Optimización de justicia y rendimiento en procesadores multicore asimétricos mediante planificación consciente de la contención

Adrián García García

Director: Juan Carlos Sáez Alcaide

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Febrero 2018
Calificación: 10

Optimización de justicia y rendimiento en procesadores multicore asimétricos mediante planificación consciente de la contención

Memoria de Trabajo Fin de Máster en Ingeniería Informática

Adrián García García

Director: Juan Carlos Sáez Alcaide

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Febrero 2018
Calificación: 10

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Autorización de difusión y utilización

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Optimización de justicia y rendimiento en procesadores multicore asimétricos mediante planificación consciente de la contención", realizado durante el curso académico 2017-2018 bajo la dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Adrián García García

Madrid, a 20 de Febrero de 2018.

Agradecimientos

Quiero agradecer en primer lugar a mi tutor, Juan Carlos Sáez Alcaide por todo el apoyo recibido y el esfuerzo puesto en realizar este trabajo.

Este trabajo ha sido financiado por la Unión Europea (FEDER) y por el Ministerio de Economía, Industria y Competitividad bajo el Proyecto de Investigación con referencia TIN 2015-65277-R, así como por la Red Europea de Excelencia HIPEAC-4.

Finalmente, me gustaría agradecer a David Koufaty de Intel Labs el habernos permitido experimentar con el prototipo QuickIA de Intel.

Publicaciones derivadas del TFM

- Adrian Garcia-Garcia, Juan Carlos Saez, Manuel Prieto-Matias. "Delivering fairness on asymmetric multicore systems via contention-aware scheduling". *5th Workshop on Runtime and Operating Systems for the Many-core Era. Euro-Par 2017: Parallel Processing Workshops. Lecture Notes in Computer Science (LNCS)* (En prensa).
- Adrián García García, Juan Carlos Sáez, Manuel Prieto. "Optimización de la justicia en procesadores multicore asimétricos mediante planificación consciente de la contención". *Actas de las Jornadas SARTECO 2017*. ISBN: 978-84-697-4835-0, pp. 255-264. (2017)

Resumen

Optimización de justicia y rendimiento en procesadores multicore asimétricos mediante planificación consciente de la contención

Los procesadores multicore asimétricos (AMPs) con repertorio común de instrucciones constituyen una alternativa de mayor eficiencia energética que los multicores simétricos para cargas de trabajo diversas. Los AMPs integran cores rápidos de alto rendimiento, con otros más lentos y de bajo consumo. Se ha demostrado que la planificación a nivel de sistema operativo y consciente de la asimetría es esencial para obtener beneficios significativos en cuanto a rendimiento global y para garantizar justicia en este tipo de sistemas. No obstante, para poder llevar esto a cabo, el planificador ha de estimar de forma precisa el progreso que cada hilo realiza al ejecutarse en los diversos tipos de core durante la ejecución.

A pesar de la existencia de planificadores que optimizan la justicia o el rendimiento en AMPs, las propuestas existentes habitualmente dependen de extensiones hardware especiales o de modelos de predicción específicos de plataforma y, además, no tienen en cuenta la degradación del rendimiento asociada a la contención en los recursos compartidos (p.ej., caché compartida o bus de memoria). Esto puede limitar la portabilidad del planificador y producir una degradación significativa de la justicia y del rendimiento global.

En este Trabajo de Fin de Máster se ha procedido al diseño e implementación en el kernel Linux de un planificador consciente de la contención en recursos compartidos en AMPs, que está orientado a la optimización de la justicia. Asimismo, el planificador expone un parámetro de configuración que permite mejorar gradualmente el rendimiento global a costa de degradar la justicia. La evaluación experimental del planificador propuesto se ha llevado a cabo utilizando hardware multicore asimétrico real.

Palabras clave: Procesadores multicore asimétricos, planificación, sistemas operativos, justicia, contención de recursos compartidos, kernel Linux.

Abstract

Fairness and throughput optimization on asymmetric multicore processors via contention-aware scheduling

Single-ISA Asymmetric Multicore Processors (AMPs) constitute a more energy efficient design approach than symmetric multicores for a wider variety of workloads. AMPs combine on the same chip high-performance big cores with energy-efficient small ones. Previous research has highlighted that asymmetry-aware OS scheduling is paramount when it comes to ensuring fairness while delivering acceptable throughput on AMP systems. Nonetheless, to accomplish this the scheduler must accurately track the progress made by each thread while running on different core types throughout the execution.

Despite the fact that some schedulers that seek to optimize fairness or throughput on AMPs have been proposed, these proposals usually rely on special hardware extensions or platform-specific estimation models to function, and, more importantly, they do not take into account the performance degradation that comes from shared-resource contention (e.g. due to shared caches/buses). This may limit the scheduler portability and may also lead to significant throughput/fairness degradation.

The goal of this Master Thesis is to overcome these issues by designing and implementing an OS-level fairness-oriented scheduler that is aware of shared resource contention on AMPs. Notably, our scheduling proposal also exposes a configurable parameter making it possible to deliver a configurable trade-off between fairness and throughput. The experimental evaluation of our scheme has been conducted on real asymmetric hardware.

Keywords: Asymmetric multicore processors, scheduling, operating systems, fairness, shared resource contention, Linux kernel.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

En los últimos años dos tendencias principales han surgido en el diseño y fabricación de microprocesadores: la integración de un mayor número de cores por chip y la incorporación de diferentes tipos de cores en la misma plataforma para un uso general o específico. La segunda tendencia ha generado un interés creciente por las arquitecturas heterogéneas debido a las posibilidades que ofrecen las configuraciones de estos sistemas. El grado de especialización divide a las arquitecturas heterogéneas en varias clases, cada una representando un punto único en el espacio de diseño [32, 27]. En un extremo de este espectro se encuentran los sistemas que combinan un número reducido de cores de alto rendimiento junto a aceleradores [9] o procesadores de propósito específico. Éste es el caso de los sistemas como el IBM Cell Broadband Engine [19] o de plataformas CPU-GPU, donde los diferentes cores típicamente exponen distintos repertorios de instrucciones (*Instruction Set Architecture* o ISA). A pesar de sus beneficios, estas arquitecturas habitualmente requieren un importante esfuerzo de desarrollo [27, 35]. Este tipo de plataformas heterogéneas contrasta con los procesadores multicore asimétricos (AMPs) con repertorio común de instrucciones [26], que integran una combinación de cores de alto rendimiento y complejos desde el punto de vista microarquitectónico (cores rápidos), con procesadores más simples y de mayor eficiencia energética (cores lentos). Este trabajo se centra en este tipo de arquitectura heterogénea.

A pesar de que una aplicación puede diseñarse específicamente para explotar las capacidades de los diferentes tipos de core de un AMP y ejecutarse asignando manualmente los diferentes hilos a cada tipo de core (p.ej., usando máscaras de afinidad), el sistema operativo necesita afrontar una serie de retos para proporcionar de forma transparente el potencial completo de los multicores asimétricos a las aplicaciones sin requerir su modificación [27, 32]. Algunos de estos retos deben de ser resueltos por el planificador [24, 38]. Un aspecto importante es cómo distribuir de forma efectiva el tiempo de ejecución en los cores rápidos entre las diferentes aplicaciones que se ejecutan en el sistema. La mayor parte de algoritmos

de planificación propuestos para multicores asimétricos han sido diseñados para optimizar el rendimiento global del sistema para cargas de trabajo multiprogramadas [26, 42, 24, 39, 44]. Para ello, el planificador debe dedicar los cores rápidos a ejecutar aquellas aplicaciones que los utilizan de forma efectiva, ya que obtienen una importante mejora del rendimiento (speedup) respecto a cuando se ejecutan en cores lentos [26]. Se pueden obtener ganancias adicionales en el rendimiento si se utilizan los cores rápidos para acelerar las fases de ejecución secuenciales y otros cuellos de botella de escalabilidad presentes en programas multihilo, empleando diferentes técnicas software [4, 39, 17] o hardware [18, 29].

No obstante, los planificadores conscientes de la asimetría que únicamente intentan optimizar el rendimiento global son inherentemente injustos [38], lo cual puede provocar una serie de consecuencias negativas en el sistema [33, 10]. Por ejemplo, el tiempo de ejecución de una aplicación puede variar significativamente entre diferentes ejecuciones, dependiendo de las otras aplicaciones de la carga de trabajo [38]. Por otra parte, aplicaciones con la misma prioridad podrían no experimentar la misma degradación del rendimiento cuando se ejecutan junto a otras en una carga de trabajo, con respecto a cuando se ejecutan solas en el AMP. Estos problemas hacen que las estrategias basadas en prioridades sean ineficaces [10]. Con ello además se dificulta ofrecer garantías de rendimiento [27, 49, 12] y se podrían realizar cargos erróneos en servicios comerciales de computación en la nube, donde se factura a los usuarios por tiempo de CPU [33].

Este Trabajo de Fin de Máster explora principalmente cómo planificar de forma justa y a nivel del sistema operativo un conjunto de aplicaciones sin modificar en un sistema multicore asimétrico. Con esta finalidad, el planificador debe igualar el progreso realizado por las distintas aplicaciones de la carga de trabajo a lo largo de su ejecución en los diferentes tipos de cores [45, 38]. Para conseguirlo, el planificador deberá estar provisto de un mecanismo que le permita conocer la degradación del rendimiento (*slowdown*) acumulada por cada aplicación en tiempo de ejecución con respecto a su ejecución sola en el sistema. En sistemas multicore asimétricos, la degradación del rendimiento depende de dos factores principales:

1. **Asimetría en rendimiento**. La mayoría de aplicaciones experimentan un beneficio significativo cuando utilizan los cores rápidos de alto rendimiento con respecto a cuando se ejecutan en cores lentos de bajo consumo. Cuando el número de hilos de la carga de trabajo excede la cantidad de cores rápidos del sistema, algunos hilos deben ejecutarse en cores lentos durante un tiempo determinado [27, 24, 39]. En este contexto, la ejecución de las aplicaciones se ralentiza de forma proporcional a su factor de ganancia o *speedup factor* (el beneficio relativo que obtendría al ejecutarse usando cores rápidos). Cabe destacar que este valor puede variar significativamente entre aplicaciones o incluso entre diferentes fases de ejecución del mismo programa [26, 7].

2. **Contención en recursos compartidos**. La contención en recursos compartidos también juega un papel importante en la degradación del rendimiento que sufren las aplicaciones en multicores asimétricos. En las plataformas multicore asimétricas disponibles en la actualidad, cada conjunto de cores del mismo tipo (rápidos o lentos) habitualmente comparte una caché de último nivel [5, 15, 8] y otros recursos de memoria con el resto de cores. Las aplicaciones que se ejecutan en los diferentes cores pueden competir entre ellas por espacio en memoria caché y por el uso de ancho de banda del bus, lo que puede provocar una degradación del rendimiento de un modo desigual e impredecible entre aplicaciones, ya que el hardware comercial actual no es capaz de garantizar por sí solo un uso justo de los diferentes recursos compartidos [46, 52, 21]. Nótese que también puede producirse contención en el controlador de DRAM compartido por todos los cores del sistema.

Recientemente se han propuesto diferentes estrategias de planificación para multicores asimétricos, tales como Equal-Progress [45] o ACFS [38], que intentan garantizar justicia teniendo en cuenta únicamente la asimetría del rendimiento. No obstante, no tienen en cuenta los efectos de la contención de recursos compartidos cuando realizan decisiones de planificación. Tal y como se demuestra en este trabajo, esto provoca una degradación sustancial del rendimiento y de la justicia cuando múltiples aplicaciones intensivas en memoria están presentes en la carga de trabajo. A pesar de que existen algunas propuestas conscientes de la contención que intentan proporcionar justicia [46, 12, 52] no están diseñadas para multicores asimétricos cuyos cores pueden exhibir importantes diferencias en rendimiento, eficiencia energética y aspectos microarquitectónicos. Por lo tanto, estas estrategias no tienen en cuenta los aspectos relacionados con la asimetría en el rendimiento.

1.1. Objetivos del proyecto y plan de trabajo

Para solventar las limitaciones de las propuestas existentes, este trabajo tiene como objetivo el desarrollo de CAMPS, un planificador a nivel del sistema operativo y consciente de la contención para sistemas multicore asimétricos, que intenta optimizar la justicia manteniendo un rendimiento global aceptable en el contexto de cargas de trabajo intensivas en cómputo de larga duración. Se tratará de avanzar el estado del arte en planificación orientada a justicia para multicores asimétricos y, al mismo tiempo, teniendo en cuenta los efectos de la contención de recursos compartidos.

Para alcanzar estos objetivos, el plan de trabajo seguido ha constado de las siguientes fases:

- Realización de un estudio preliminar para evaluar el impacto que tiene la contención en recursos compartidos en diferentes plataformas asimétricas comerciales.

- Diseño de un mecanismo que, basándose en las conclusiones obtenidas en el estudio preliminar, permite predecir la degradación del rendimiento que cada hilo de la carga de trabajo experimenta al ejecutarse en los diferentes tipos de core del AMP. Este mecanismo de predicción requiere la monitorización en tiempo de ejecución de diferentes métricas del rendimiento disponibles en las plataformas asimétricas comerciales.

- Implementación del algoritmo como una clase de planificación del kernel Linux que es capaz de garantizar la justicia mientras se mantiene un rendimiento global aceptable en el AMP.

- Incorporación en el planificador de un parámetro que permita al administrador del sistema (1) mejorar gradualmente el rendimiento global a costa de empeorar la justicia, o bien (2) optimizar únicamente el rendimiento global del sistema.

- Evaluación experimental de la propuesta de planificación utilizando hardware multicore asimétrico real.

1.2. Estructura de la memoria

El resto de la memoria se estructura de la siguiente forma:

- En el **capítulo 2** se introduce el concepto de justicia empleado en el trabajo y se discuten los desafíos asociados al cálculo de la degradación del rendimiento de cada hilo en tiempo de ejecución. También se presentan los resultados de un estudio experimental que permite evaluar los efectos de la contención en recursos compartidos en multicore asimétricos.

- En el **capítulo 3** se expone cómo se puede obtener el factor de ganancia o SF en tiempo de ejecución y se habla de diferentes estrategias que utilizan este factor para optimizar el rendimiento global. Más adelante, se discuten diferentes propuestas que tratan de garantizar la justicia en multicores asimétricos. Por último, se presentan diferentes propuestas que tienen en cuenta la contención en recursos compartidos.

- En el **capítulo 4** se proporciona una idea general de la implementación de CAMPS en el kernel Linux. Tras esto, se pasa a describir el mecanismo usado por el *performance monitor* para predecir dinámicamente el factor de degradación del rendimiento de un hilo y se explica la técnica de intercambio de hilos utilizada por el *core scheduler* para garantizar la justicia. A continuación, se presenta el modo *non-work-conserving* (NWC), que entra en funcionamiento en situaciones específicas para predecir de forma más precisa el factor de degradación del rendimiento de ciertos hilos. Finalmente, se explican los mecanismos específicos implementados en el planificador para tratar de una forma efectiva con aplicaciones multihilo y, también, se describe el mecanismo usado por CAMPS para intercambiar justicia por rendimiento global.

- En el **capítulo 5** se comienza realizando una comparativa de la efectividad de CAMPS con respecto a otros planificadores conscientes de la asimetría propuestos con anterioridad [7, 24, 45, 38]. Tras esto, se ilustra la efectividad que tiene el mecanismo incluido en CAMPS para intercambiar justicia por rendimiento global.

- En el **capítulo 6** se presentan las conclusiones obtenidas de este trabajo y se proponen posibles avenidas de trabajo futuro.

- Finalmente, se proporcionan varios apéndices. En ellos se incluye: (A) Introducción y (B) Conclusiones del Trabajo de Fin de Máster traducidas al inglés.

Capítulo 2

Motivación

En este capítulo se proporcionan detalles sobre las plataformas experimentales utilizadas en este trabajo, se introduce el concepto de justicia empleado y se discuten los desafíos asociados al cálculo de la degradación del rendimiento de cada hilo en tiempo de ejecución. Finalmente, se presentan los resultados de un estudio experimental que ilustra la principal observación que explotamos para determinar la degradación del rendimiento dinámicamente en multicore asimétricos.

2.1. Plataformas experimentales

Para realizar nuestro análisis experimental empleamos tres sistemas multicore asimétricos: la placa de desarrollo ARM Juno [5], que integra un procesador big.LITTLE de ARM de 64 bits; la placa Odroid XU-4 [15], equipada con un procesador big.LITTLE de 32 bits; y el prototipo QuickIA[1] de Intel [8], que consta de un procesador Xeon de alto rendimiento, y un procesador Atom de bajo consumo. La tabla 2.1 resume las especificaciones de los sistemas empleados. En la figura 2.1 se muestra una imagen de ellos.

2.2. Justicia en sistemas multicore asimétricos

Existen múltiples trabajos previos sobre justicia en procesadores multicore simétricos [13, 10] y asimétricos [18, 45], que definen un planificador como justo si aquellas aplicaciones de una carga de trabajo que tienen la misma prioridad experimentan la misma degradación del rendimiento o *slowdown* por el hecho de

[1]Este prototipo ha sido donado recientemente por Intel Labs (Hillsboro, OR, USA) a la Facultad de Informática de la UCM.

Tabla 2.1: Características de los sistemas AMP empleados en este trabajo

Sistema	Placa ARM Juno		Placa Odroid XU4		Prototipo QuickIA de Intel	
Tipos de core	Cortex A57	Cortex A53	Cortex A15	Cortex A7	Xeon E5450	Atom N330
Núm. cores	2	4	4	4	$4^{(*)}$	2
Frecuencia	1.10GHz	850 MHz	2.0 Ghz	1.4 Ghz	1.2 Ghz	1.6 Ghz
Pipeline	Fuera de orden	En orden	Fuera de orden	En orden	Fuera de orden	En orden
Caché L2	2MB/16-vías	1MB/16-vías	2MB/16-vías	512KB/8-vías	6MB/16-vías	512KB/8-vías
DRAM	8GB DDR3 @ 800MHz		2GB DDR3 @ 933MHz		16GB DDR2 @ 677MHz	

(*) En nuestros experimentos desactivamos dos cores de alto rendimiento, para lograr una topología similar a la de los sistemas de ARM.

Figura 2.1: Placa Juno, placa Odroid XU4 y prototipo QuickIA de Intel (respectivamente)

compartir el sistema. Según esta definición, para cuantificar la justicia empleamos la métrica *unfairness* (cuanto menor mejor), que ha sido ampliamente utilizada en trabajos anteriores [13, 33, 10, 46, 18, 38]. Esta métrica se define de la siguiente forma:

$$Unfairness = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)} \tag{2.1}$$

siendo n el número de aplicaciones de la carga de trabajo y $Slowdown_i = \frac{CT_{sched,i}}{CT_{alone,i}}$. A su vez, $CT_{sched,i}$ denota el tiempo de ejecución de la aplicación i usando un algoritmo de planificación determinado, y $CT_{alone,i}$ es el tiempo de ejecución de la aplicación i cuando se ejecuta sola en el sistema AMP (con todos los cores rápidos disponibles).

La degradación del rendimiento que experimenta un hilo o una aplicación monohilo durante una fase de ejecución específica puede calcularse usando el número de instrucciones por segundo (IPS) de la siguiente manera:

$$Slowdown = IPS_{alone}/IPS_{sched} \tag{2.2}$$

siendo IPS_{alone} el número de instrucciones por segundo para la fase de ejecución cuando se ejecutó solo en el sistema, e IPS_{sched} denota el IPS alcanzados por el hilo cuando ejecuta la misma fase en el contexto de una carga de trabajo multiprogramada bajo un algoritmo de planificación específico.

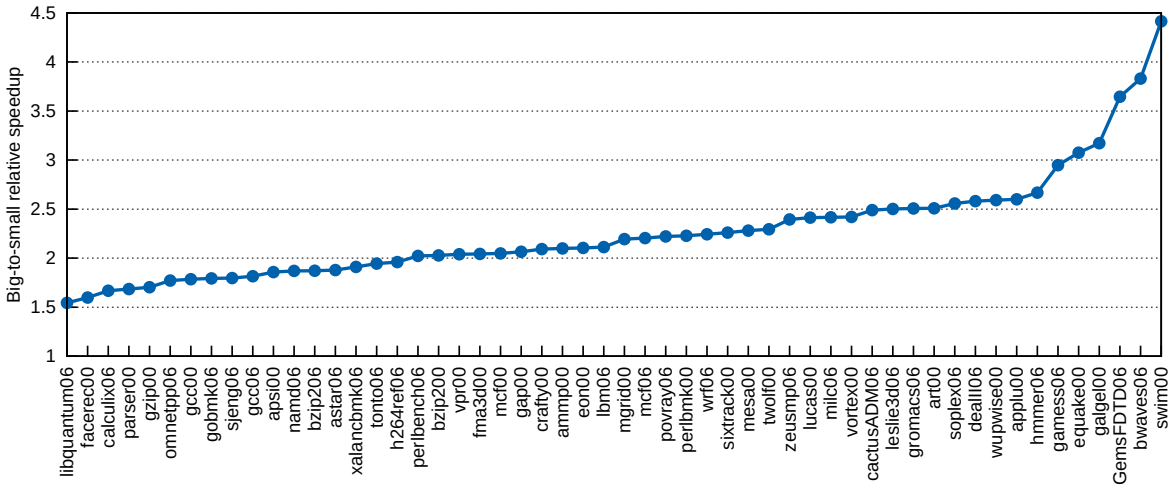


Figura 2.2: Factor de ganancia medio que experimentan diferentes aplicaciones de la suite SPEC CPU cuando se ejecutan en un core rápido respecto a uno lento en la placa ARM Juno. Nótese que el sufijo ("00" o "06") sirve para diferenciar su pertenencia a CPU2000 o CPU2006.

En este trabajo de fin de máster se asume que el IPS_{alone} en un sistema asimétrico se maximiza cuando el hilo se ejecuta en un core de alto rendimiento (rápido) en aislamiento; éste es el caso de todas las aplicaciones exploradas para las diferentes plataformas experimentales usadas en este trabajo. Cabe destacar que en el contexto de las aplicaciones multihilo, la métrica IPS puede ser un indicador engañoso del rendimiento, ya que un hilo puede exhibir valores altos de IPS mientras realiza una espera activa (*spin*) en una primitiva de sincronización (p.ej. barrera). Para conseguir que el planificador del sistema operativo sea consciente de estas situaciones, donde los hilos no realizan trabajo útil, en este trabajo explotamos las notificaciones de spin desde el *runtime system*, que se ejecuta en espacio de usuario. Este aspecto se describe en detalle en la sección 4.5.

Para ilustrar el hecho de que no todas las aplicaciones obtienen el mismo beneficio al ejecutarse en diferentes tipos de core de un procesador asimétrico, la figura 2.2 muestra el factor de ganancia medio observado para aplicaciones de las suites de benchmarks SPEC CPU2000 y SPEC CPU2006 al ejecutarse en aislamiento en un core rápido respecto a uno lento en la placa Juno de ARM [5], que contiene un procesador big.LITTLE de ARM de 64 bits formado por dos cores *big* Cortex A57 y cuatro cores *small* Cortex A53. Para benchmarks secuenciales como estos, el beneficio relativo de la aplicación es el mismo que el factor de ganancia (*speedup factor* o SF) del único hilo, definido como $\frac{IPS_{big}}{IPS_{small}}$, donde IPS_{big} y IPS_{small} son las tasas de instrucciones por segundo (IPS) que se obtienen cuando el hilo se ejecuta

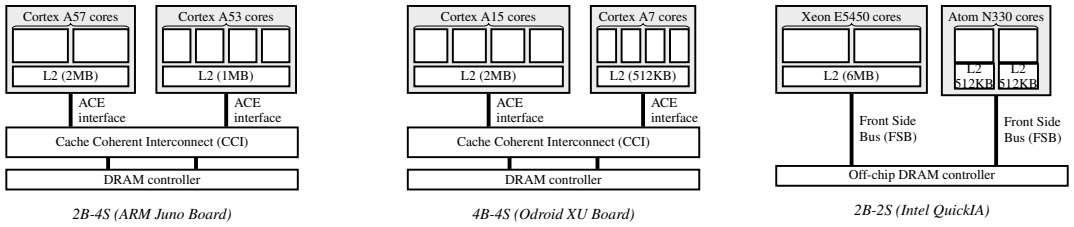


Figura 2.3: Configuraciones de las plataformas asimétricas

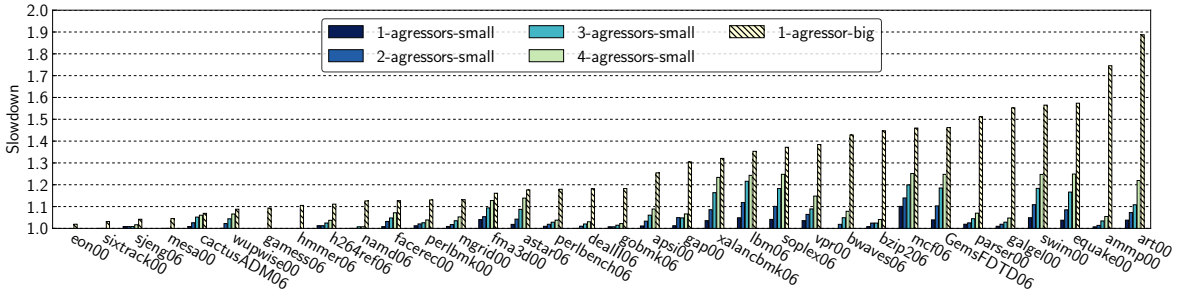
solo respectivamente en cores rápidos y lentos. Como se puede apreciar, todas las aplicaciones obtienen incrementos en rendimiento cuando se ejecutan en un core rápido respecto a uno lento ($speedup > 1$) y este hecho debería ser tenido en cuenta por un planificador que intente garantizar justicia.

En el ámbito de la planificación en procesadores multicore asimétricos se considera un scheduler como justo [10, 46, 38, 45] si es capaz de garantizar que todos los hilos que se ejecutan simultáneamente experimentan la misma degradación del rendimiento relativa o *slowdown* por el hecho de compartir el sistema a lo largo de su ejecución, y manteniendo al mismo tiempo un rendimiento aceptable. Para conseguirlo, el planificador debe contar con un mecanismo para determinar la degradación del rendimiento de un hilo en tiempo de ejecución.

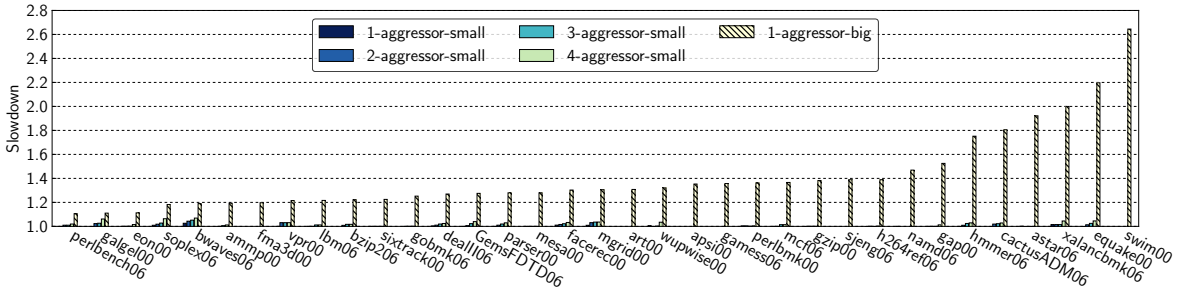
Cabe destacar que, determinar la degradación del rendimiento dinámicamente con la ecuación 2.2 es complicado en la práctica; aunque IPS_{sched} puede obtenerse fácilmente mediante contadores hardware, averiguar con precisión IPS_{alone} en tiempo de ejecución es una tarea compleja, incluso en CMPs simétricos [43, 52]. Para solventar este problema, los algoritmos de planificación para CMPs simétricos habitualmente utilizan modelos de estimación para aproximar los IPS_{alone} [46], o emplean diferentes heurísticas para determinar el grado de degradación del rendimiento indirectamente a través de métricas relacionadas con la contención [52], como la tasa de fallos de caché de último nivel (LLC) [51, 46].

2.2.1. Impacto de la contención de recursos compartidos en AMPs

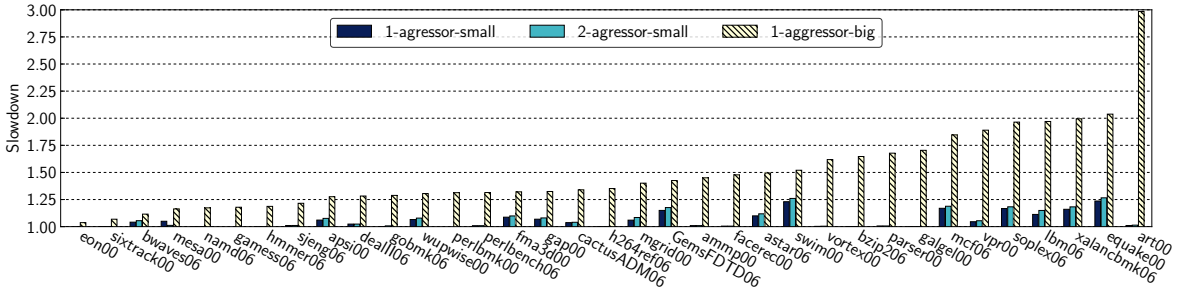
Algunos planificadores orientados a justicia en AMPs propuestos recientemente [45, 38], asumen que la degradación del rendimiento por contención sufrida por un hilo (relativa a su ejecución en aislamiento) es despreciable cuando se ejecuta en un core rápido, incluso aunque comparta ese conjunto de cores (cluster) con otros hilos. De esta forma, la degradación del rendimiento de un hilo se estima como 1 cuando se ejecuta en un core rápido. Además, el factor de ganancia relativo



(a) Placa Juno de ARM (Cortex A53 + Cortex A57 cores)



(b) Placa Odroid XU4 (Cortex A7 + Cortex A15 cores)



(c) Prototipo QuickIA de Intel (Atom + Xeon)

Figura 2.4: Degradación del rendimiento (slowdown) relativa a la ejecución sola en el sistema de diferentes benchmarks SPEC CPU con respecto a su ejecución simultánea con varias instancias de una aplicación agresora. Nótese que el sufijo (“00” o “06”) sirve para diferenciar su pertenencia a la versión CPU2000 o CPU2006.

de un hilo –conocido como *speedup factor* (SF) [39]– se usa para aproximar la degradación del rendimiento cuando el hilo se ejecuta en un core lento. El SF puede determinarse en tiempo de ejecución de diferentes formas, que se discutirán más adelante en la sección 3.

No obstante, asumir que la degradación del rendimiento de un hilo cuando se ejecuta en un core rápido es despreciable (como se hace en [45, 38]) es poco realista en escenarios en los que los hilos compiten entre sí de forma intensiva por los recursos compartidos. Para cuantificar el impacto que tiene la contención en el rendimiento, medimos el grado de degradación del rendimiento sufrido por diferentes aplicaciones mientras están asignadas a un core rápido al mismo tiempo que un número cada vez mayor de instancias de una aplicación agresora. En este estudio, se han usado benchmarks con distinto tamaño del conjunto de trabajo y *memory footprint* de las suites SPEC CPU2000 y SPEC CPU2006 [14]. Como aplicación agresora, se ha elegido el benchmark `bandwidth` [50], que genera una contención significativa en el último nivel de caché, los buses compartidos y el controlador de DRAM. En las plataformas evaluadas, se observó que `bandwidth` causa un mayor nivel de contención que otros benchmarks que hacen un uso muy intensivo de memoria, como `lbm` [43, 49] de la suite SPEC CPU.

Para nuestro estudio experimental se han empleado dos plataformas ARM equipadas con procesadores big.LITTLE de 32 y 64 bits respectivamente –la placa Juno de ARM [5] y la placa Odroid XU4 [15]. También se ha experimentado con el prototipo QuickIA de Intel [8]. En estas plataformas asimétricas, que integran una combinación de cores de alto rendimiento (con ejecución fuera de orden) y cores de bajo consumo (con ejecución en orden), los benchmarks SPEC CPU exhiben un amplio rango de factores de ganancia: 1.55x-4.44x (Juno), 1.36x-6.63x (Odroid) y 1.02x-4.7x (QuickIA). La figura 2.3 muestra la organización de la jerarquía de memoria, así como el número de cores de cada tipo, de las diferentes configuraciones AMP exploradas. Por mayor simplicidad, se emplea la notación nB-mS para referirnos a cada configuración asimétrica, dónde n y m denotan el número de cores rápidos (Big) y lentos (Small), respectivamente. Notablemente, en las configuraciones 2B-4S y 4B-4S el conjunto de cores del mismo tipo (rápidos o lentos), que forma un cluster, comparte el último nivel de caché (L2) y una interfaz al bus (AMBA). En la plataforma 2B-2S (un sistema de doble socket), la interfaz de bus (FSB) también es compartida entre los cores del mismo tipo, pero la caché de último nivel solo se comparte en el conjunto de cores big. Además, todas las plataformas tienen un único controlador de DRAM compartido entre todos los cores sin importar su tipo.

La figura 2.4 muestra la degradación del rendimiento (relativa a la ejecución aislada) que experimentan diferentes aplicaciones al ejecutarse simultáneamente con varias instancias de la aplicación agresora `bandwidth`. Para cada benchmark, que siempre se ejecuta en un core rápido en los experimentos, se exploraron diferentes escenarios. En el primero, denominado como *1-aggressor-big* en la figura 2.4, el benchmark se ejecuta al mismo tiempo con una instancia de `bandwidth`, que está asignado también a un core rápido; los cores lentos permanecen *idle* en este caso. En el resto de escenarios, etiquetados como *N-aggressors-small*, *N* instancias de `bandwidth` se asignan a los cores lentos; así, al dejar el resto de cores rápidos sin usar, se elimina la contención en el último nivel de caché (LLC) y la interfaz del bus asociada con el cluster de cores rápidos, pero no en el controlador de DRAM.

Como se observa en los resultados, la degradación del rendimiento puede ser significativa (hasta 1.89x en la plataforma Juno, hasta 2.65x en la placa Odroid, y hasta 2.98x el prototipo QuickIA) cuando tanto el benchmark como la instancia de la aplicación agresora se ejecutan simultáneamente en los cores rápidos, incluso aunque los cores lentos estén sin usar. En contraposición, si un benchmark agresor se ejecuta en un core lento la degradación del rendimiento baja significativamente; la mayor parte de los benchmarks muestran una degradación del rendimiento inferior al 10 % en este caso. Cabe destacar que, si se ocupan todos los cores lentos disponibles con instancias de la aplicación agresora, la degradación no supera el 26 %, valor muy inferior al que se produce cuando el agresor se ejecuta en un core rápido. Se consideran dos las principales causas de este comportamiento. Por una parte, la contención en la LLC y el bus compartido (en el cluster de cores rápidos) se elimina completamente en el caso de *N-aggressors-small*. Por otra parte, se ha observado que la presión que un benchmark agresor ejerce en la memoria compartida es mayor cuando se ejecuta en un core rápido en lugar de uno lento. Esto tiene que ver con el hecho de que los core lentos con ejecución en orden no pueden gestionar múltiples fallos de caché de forma simultánea, por lo que sufren bloqueos con mayor frecuencia por accesos a memoria y realizan una menor utilización del bus y del ancho de banda con memoria. De esta forma, generan un menor grado de contención.

Las principales conclusiones de este estudio preliminar son las siguientes:

- La contención en recursos compartidos puede ser significativa, por lo que deber de ser tenida en cuenta a la hora de monitorizar la degradación del rendimiento de un hilo para mantener la justicia y garantizar una utilización efectiva de los cores rápidos. Nótese además que, para algunos benchmarks, el beneficio obtenido al ejecutarse en un core rápido respecto a uno lento podría ser reducido notablemente debido a la alta degradación del rendimiento causada por la contención en recursos compartidos. Por ejemplo, esto es lo que

sucede con el benchmark `bzip2` en la placa Juno, o el programa `xalancbmk` en el prototipo QuickIA de Intel, que derivan un factor de ganancia relativo de 1.8 y 2.75 respectivamente, mientras que sufren una degradación sustancial del rendimiento derivada de la contención.

- Como los resultados revelan, la degradación del rendimiento que un hilo puede sufrir en un core rápido debido a la interferencia con hilos intensivos en memoria asignados a cores rápidos (hasta 2.98x en el prototipo QuickIA de Intel) es mucho mayor que la degradación que surge cuando se ejecutan múltiples benchmarks agresores en los cores lentos (de hasta 1.26x, alcanzada con tantos benchmarks agresores como cores lentos). Lo que es más, esta degradación del rendimiento es bastante reducida si la comparamos con los factores de ganancia relativos observados para los benchmarks SPEC CPU –como se mencionó anteriormente, el factor de ganancia puede ser de hasta 6.63x–. Esta observación sugiere que monitorizar las instrucciones por segundo (IPS) de un hilo cuando se ejecuta en un core rápido en una situación de baja contención en el cluster de cores rápidos (p.ej., con el resto de cores inactivos) podría ser una buena estimación de IPS_{alone}. El planificador propuesto en este TFM explota esta observación para aproximar la degradación del rendimiento, como se describe en la sección 4.

- Algunas aplicaciones no experimentan una degradación del rendimiento notable debido a la contención cuando se ejecutan compartiendo el cluster de cores con otras aplicaciones intensivas en memoria como `bandwidth`. Por ejemplo, esto ocurre con los benchmark `sixtrack`, `eon` o `mesa`. Como se ha demostrado en trabajo previo [51, 43, 46], las aplicaciones intensivas en CPU con un conjunto de trabajo pequeño (que se puede alojar en un espacio reducido de la LLC) y una buena localidad de caché, o aquellas que no usan la jerarquía de memoria de forma intensiva, no experimentan una relevante penalización del rendimiento debido a la contención. De forma similar a [46], el planificador propuesto en este trabajo utiliza el Bus Transfer Rate (BTR) para identificar aquellas situaciones en las que los hilos tienen una menor probabilidad de sufrir contención cuando se ejecutan en un cluster de cores rápidos. En las plataformas utilizadas, el BTR se mide de la siguiente forma:

$$\frac{accesos\_lectura\_bus * tam\_linea\_cache * frecuencia\_procesador}{numero\_total\_ciclos}$$

Capítulo 3

Trabajo relacionado

En la actualidad, existe una gran cantidad de trabajos que muestran los potenciales beneficios de usar los procesadores multicore asimétricos frente a sus respectivos simétricos [25, 4]. No obstante, los sistemas AMP plantean una serie de retos significativos para el planificador del sistema operativo [32, 27]. Garantizar la justicia mientras se mantiene un rendimiento global aceptable supone un importante reto, que es el principal objeto de estudio de este trabajo.

Investigaciones recientes han señalado que la justicia, el rendimiento global del sistema y la eficiencia energética representan objetivos de optimización contrapuestos en AMPs [41]. En concreto, al optimizar la justicia se produce habitualmente una degradación significativa del rendimiento y la eficiencia energética.
En general, para optimizar cualquiera de los objetivos enumerados, el planificador debe tener en cuenta el factor de ganancia o *speedup factor* (SF) de los diferentes hilos a la hora de tomar decisiones de planificación [24, 44, 39, 38].

Este capítulo se estructura de la siguiente forma, en primer lugar se habla de cómo se puede obtener el factor de ganancia o SF en tiempo de ejecución y se habla de diferentes estrategias que utilizan este factor para optimizar el rendimiento global. Más adelante, se discuten diferentes propuestas que tratan de garantizar la justicia en los multicores asimétricos. Por último, se presentan diferentes propuestas que tienen en cuenta la contención en recursos compartidos.

3.1. Obtener el SF en tiempo de ejecución

Para maximizar el rendimiento gobal en el contexto de cargas de trabajo multiprogramadas, se ha demostrado en investigaciones previas que se debe seguir la aproximación HSP (*High Speedup*), que se basa en ejecutar de forma prioritaria en los cores rápidos aquellas aplicaciones que derivan un mayor factor de ganancia (SF). La principal diferencia entre las diferentes variantes de la aproximación HSP se centra en el mecanismo empleado para obtener los factores de ganancia de los hilos en tiempo de ejecución.

Hasta la fecha, se han utilizado tres técnicas para obtener el SF de un hilo dinámicamente. La primera técnica se basa en medir el SF de forma directa [26, 7], lo que conlleva ejecutar cada hilo tanto en un core rápido como en uno lento para poder obtener las IPC de ambos tipos de core. En trabajos previos se ha demostrado que esta aproximación, también conocida como *IPC sampling*, está sujeta a variabilidades que pueden surgir al utilizar para la aproximación del SF valores de IPC pertenecientes a diferentes fases de ejecución [42]. La segunda técnica se basa en el uso de un modelo de predicción de SF que se alimenta de diferentes métricas de rendimiento recogidas en el tipo de core actual mediante contadores hardware [24, 39, 36, 38]. La principal limitación de esta aproximación es que requiere la construcción de un modelo de estimación específico de plataforma asimétrica. Esto obliga a realizar un análisis previo que permita identificar los conjuntos de eventos hardware disponibles en cada plataforma y que sean relevantes para la predicción del SF [24, 39]. Además, es necesario calcular los valores de los diferentes coeficientes del modelo si este se basa en técnicas de regresión([36, 39, 38]). La tercera aproximación se conoce como Performance Impact Estimation (PIE) [44], un mecanismo hardware que es capaz de estimar el SF de forma precisa. Aunque la incorporación de soporte hardware especial es una vía de investigación que podría traer importantes beneficios, PIE presenta importantes limitaciones prácticas que complican su integración en hardware asimétrico real [36].

La propuesta de planificación de este trabajo (CAMPS) aproxima el ratio de rendimiento relativo midiendo el IPS en el core actual, y comparándolo con una estimación de IPS_{alone} (aproximado con los valores de IPS medido en cores rápidos durante diferentes fases de ejecución en escenarios de baja contención). Esta forma de calcular SF nos permite evitar las imprecisiones relacionadas con la toma de muestras de diferentes fases de ejecución de las técnicas basadas en IPC sampling [42], y hace posible tener en cuenta la alta variabilidad del IPS que se produce en situaciones de alta contención en recursos compartidos. Nótese que al emplear esta estrategia, CAMPS no depende de modelos de estimación de SF específicos a una plataforma. Por el contrario, requiere la recolección de un conjunto fijo de métricas de rendimiento de alto nivel (las mismas para todas las plataformas), como se

explica en el capítulo 4. De esta forma se mejora la portabilidad del planificador, eliminando por completo la necesidad de realizar estudios experimentales exhaustivos para cada sistema con el objetivo de construir un modelo de estimación de SF.

3.2. La justicia en multicores asimétricos

La primera propuesta de planificación orientada a justicia para AMPs fue una versión de Round-Robin (RR), que simplemente realiza migraciones periódicas para asegurase de repartir de forma justa el tiempo de uso de cores rápidos entre las aplicaciones [7]. Este algoritmo se puede implementar de forma sencilla en la mayor parte de sistemas operativos de propósito general y no requiere soporte hardware especial. No obstante, en [28] se ha explorado una implementación basada en hardware de este esquema. Se ha demostrado que al repartir de forma equitativa el tiempo de cores rápidos se permite obtener un mayor rendimiento y unos tiempos de ejecución con menor variabilidad entre diferentes ejecuciones en AMPs [40, 27], que los planificadores por defecto de sistemas operativos de propósito general (que no tienen en cuenta la asimetría). Por este motivo, Round-Robin ha sido utilizado de forma generalizada como algoritmo de referencia para evaluar las prestaciones de planificadores orientados a justicia [7, 40, 23]. Cabe destacar que, Round-Robin representa una aproximación subóptima para garantizar la justicia [38], ya que no tiene en cuenta los diferentes beneficios relativos o SF de cada hilo, a la hora de repartir el tiempo de core rápido entre aplicaciones.

Hasta la fecha, ACFS [38] es el algoritmo de planificación orientado a justicia para sistemas multicore asimétricos que proporciona mejores resultados. En [38] los autores demostraron mediante diferentes experimentos que este planificador ofrece un mayor rendimiento y justicia que otros planificadores conscientes de la justicia, como RR [7], Equal-Progress [45] o A-DWRR [27], para un conjunto extenso de cargas de trabajo ejecutadas en hardware asimétrico real. Para optimizar la justicia, ACFS recopila valores de SF de cada hilo de la carga de trabajo para registrar continuamente el progreso relativo que hacen en el sistema asimétrico, y garantiza la justicia igualando la degradación de rendimiento (slowdown) acumulada por las diferentes aplicaciones. El SF de un hilo se calcula dinámicamente mediante un modelo de predicción que requiere los valores de diferentes métricas de rendimiento. La principal limitación de ACFS [38] (también de otras propuestas [45, 27]) viene determinada porque el planificador no tiene en cuenta los efectos de la contención de recursos compartidos. Como se demuestra en los experimentos

de este trabajo, ignorar la degradación causada por la contención provoca que el planificador exhiba un comportamiento injusto cuando múltiples aplicaciones intensivas en memoria forman parte de la carga de trabajo. En cambio, CAMPS es capaz de reconocer estas situaciones y mejorar de forma efectiva la justicia.

También en el ámbito de planificación consciente de la justicia para multicores asimétricos, Kim y Huh [23] proponen un singular método que no intenta minimizar la métrica unfairness sino que busca reducir la degradación del rendimiento sufrida por aplicaciones individuales en una carga de trabajo. En su propuesta, la degradación se mide con respecto al rendimiento obtenido bajo un planificador que reparte de forma justa el uso de cores rápidos entre las aplicaciones (de forma similar a Round-Robin). Esto se contrapone a medir la degradación del rendimiento relativa a ejecutar la aplicación sola en el sistema (slowdown), tal como se realiza en este trabajo y en múltiples investigaciones previas [10, 38, 45, 13]. Si se mide la degradación del rendimiento con respecto a un planificador específico se podría estar ignorando el impacto de la contención de recursos compartidos, que supone un factor relevante para evaluar la efectividad de los diferentes planificadores. Por lo cual, en este trabajo se emplea la noción de justicia más común que se presenta en la sección 2.2.

3.3. Planificación consciente de la contención en AMPs

Hasta el momento, los únicos algoritmos de planificación conscientes de la contención para multicores asimétricos son aquellos propuestos en [11] y [6]. Cabe destacar que, a diferencia de la propuesta realizada en este trabajo, que está implementada como una nueva clase de planificación en el kernel Linux, estos planificadores están implementados como un prototipo de planificación en espacio de usuario.

Fan et al. [11] proponen un planificador para multicores asimétricos que intenta mejorar el rendimiento global en el contexto de cargas de trabajo formadas por aplicaciones secuenciales. Sin embargo, esta aproximación depende de dos modelos de estimación –*específicos para cada aplicación*– que aproximan la degradación que que sufre una aplicación por la contención de recursos compartidos. Para generar estos modelos de predicción para cada plataforma y aplicación, se requiere realizar un proceso de entrenamiento que obliga a ejecutar 80 cargas de trabajo en las que se incluye la aplicación concreta. Por el contrario, el planificador que se propone en este trabajo –principalmente diseñado para optimizar la justicia más que el rendimiento global– no necesita ningún modelo de predicción específico por cada aplicación y plataforma, por lo que se evita al usuario el árduo proceso de entre-

namiento y, al mismo tiempo, se mejora la portabilidad del planificador. Además, a diferencia de [11], CAMPS no requiere conocer con anterioridad los factores de ganancia para cada aplicación. Basarse en información estática específica a cada aplicación que se recopila en ejecuciones previas es una aproximación poco práctica para planificadores de sistemas operativos de propósito general.

En [6], Barati y otros proponen un planificador a nivel de usuario diseñado para multicores asimétricos en los que los cores se diferencian únicamente en la frecuencia de trabajo. En el contexto de procesadores que tienen cores que funcionan a diferente frecuencia pero con una igual microarquitectura se considera suficiente con tener en cuenta el grado de intensidad en memoria de cada aplicación para aproximar su degradación del rendimiento [42, 38]. Debido a esto, basarse exclusivamente en la tasa de acceso de memoria (como en [6]) podría ser efectivo en casos de asimetría basada en la frecuencia [24]. Cabe destacar que en investigaciones previas [24, 39] se ha demostrado que esta forma de asimetría en rendimiento es muy diferente a la existente actualmente en hardware AMP comercial, donde los diversos tipos de core exhiben importantes diferencias en sus características microarquitectónicas y tamaños de caché. En este contexto, existen otros aspectos más allá de la intensidad en memoria de una aplicación que deben considerarse para una planificación efectiva consciente de la justicia [24, 38]. A diferencia de [6], la aproximación en la que se basa este trabajo –implementado a nivel del sistema operativo en lugar de usuario– no hace ninguna suposición sobre la forma de asimetría en el rendimiento de la plataforma AMP. Esto nos permite realizar una amplia comparativa experimental con propuestas recientes de planificación orientadas a justicia [45, 38] y, además, empleando hardware asimétrico real.

Capítulo 4

Diseño

El algoritmo de planificación CAMPS está formado básicamente por dos elementos: el *core scheduler* y el *performance monitor*.

El *core scheduler* mantiene un contador acumulativo del progreso de cada hilo para registrar la degradación del rendimiento que experimentan a lo largo del tiempo. Para garantizar un progreso equitativo entre aplicaciones, el planificador realiza migraciones periódicas entre cores basándose en los diferentes valores de los contadores de progreso.

El *performance monitor* utiliza los contadores hardware del procesador para recabar periódicamente los valores de diferentes métricas de rendimiento para los hilos que se ejecutan en el sistema, y proporciona al *core scheduler* los datos que requiere para su correcto funcionamiento (p.ej., para estimar la degradación del rendimiento de un hilo). En particular, los valores de estas métricas pueden variar dependiendo del grado actual de contención en los recursos compartidos en la plataforma, de las características de la fase de ejecución actual de hilo o del tipo de core donde se ejecuta.

Este capítulo se estructura de la siguiente forma, en primer lugar se proporciona una idea general de la implementación de CAMPS en el kernel Linux. Tras esto, se pasa a describir el mecanismo usado por el *performance monitor* para predecir dinámicamente la degradación del rendimiento de un hilo y se explica la técnica de intercambio de hilos utilizada por el *core scheduler* para garantizar la justicia. A continuación, se presenta el modo *non-work-conserving* (NWC), que entra en funcionamiento en situaciones específicas para permitir al planificador predecir de forma más precisa la degradación del rendimiento de ciertos hilos. Finalmente, se explican los mecanismos específicos implementados en el planificador para tratar de una forma efectiva con aplicaciones multihilo y, también, se describe el mecanismo usado por CAMPS para aumentar el rendimiento global a costa de degradar la justicia.

4.1. CAMPS en el kernel Linux

El planificador del kernel Linux está equipado con múltiples algoritmos de planificación (RR, FIFO, CFS, etc.), que están implementados como clases de planificación independientes [30]. El *core scheduler* de CAMPS está implementado como una nueva clase de planificación basada en el planificador por defecto de Linux (Completely Fair Scheduler o CFS). Por otra parte, el *performance monitor* se encuentra implementado dentro de un módulo del kernel, en forma de módulo de monitorización de la herramienta PMCTrack [37]. Este diseño permite la creación de una implementación del *core scheduler* independiente de arquitectura, mientras que el código específico de plataforma para acceder a los contadores hardware y la configuración de eventos está desacoplado del código del planificador, que reside en el propio kernel [37].

Cabe destacar que el algoritmo CFS de Linux no es consciente de la asimetría en rendimiento presente en un AMP; este planificador puede asignar una aplicación de forma aleatoria a diferentes tipos de core en consecutivas ejecuciones de la misma carga de trabajo, lo que produce un comportamiento impredecible y tiempos de ejecución muy variables al lanzar una aplicación múltiples veces en el sistema AMP. Además, CFS tampoco gestiona de forma efectiva la contención en recursos compartidos [53] y no incorpora ningún mecanismo para registrar de forman precisa el progreso que un hilo hace cuando se ejecuta en diferentes tipos de core a lo largo del tiempo. De hecho, para CFS un *tick* de reloj consumido en un core rápido equivale a un *tick* consumido en un core lento. En consecuencia, CFS no garantiza, a diferencia de CAMPS, tiempos de ejecución repetibles y un progreso similar (justicia) de las diferentes aplicaciones en multicores asimétricos.

La clase de planificación de CAMPS solo se basa en la funcionalidad del planificador por defecto de Linux para llevar a cabo dos tareas: (1) equilibrar la carga entre los cores del mismo tipo, y (2) multiplexar la utilización de CPU entre los hilos asignados al mismo core (p.ej., el algoritmo de CFS se aplica cuando hay más de dos hilos asignados al mismo core). Por otra parte, el *core scheduler* de CAMPS se ocupa de garantizar el equilibrio de la carga a nivel del sistema, e intenta equilibrar el progreso relativo entre aplicaciones asignando los hilos a diferentes tipos de core y realizando migraciones entre cores si son necesarias. Ya que CAMPS está construido en base a CFS, mantiene colas por CPU de hilos listos para ejecutar. Además, CAMPS utiliza dos listas enlazadas de hilos en que se ejecutan en cores rápidos y lentos respectivamente (cada lista tiene asociado un *spinlock* de lectura y escritura).

Finalmente, cabe destacar que, a pesar del hecho que CAMPS está implementado en el kernel Linux, no existe ninguna limitación que impida que este algoritmo pueda implementarse sobre otro planificador de propósito general a nivel del sistema operativo.

4.2. Determinar dinámicamente la degradación del rendimiento

El *performance monitor* estima la degradación del rendimiento (slowdown) actual de un hilo utilizando la ecuación 2.2; el IPS actual se mide mediante contadores hardware, y el IPS_{alone} se predice empleando una tabla de historia que se mantiene para cada hilo en ejecución. Esta tabla de historia almacena los valores de IPS medidos en anteriores fases de ejecución en las que el hilo estaba asignado a un core rápido en una situación de baja contención. Como se demuestra en el capítulo 2, cuando un hilo se ejecuta en un conjunto de cores rápidos, la degradación del rendimiento que proviene de la interferencia de hilos que se ejecutan en el otro conjunto de cores es habitualmente muy pequeña. Debido a este hecho, los valores de IPS almacenados en la tabla de historia, que se han registrado cuando el hilo se ejecutaba en un core rápido en situacion de baja contencion, se utilizan para aproximar el IPS_{alone}.

Con la finalidad de detectar las situaciones de baja contención en el conjunto de cores rápidos, el planificador utiliza una heurística que se basa en la métrica *bus transfer rate* (BTR) utilizada en [47, 46]. Fundamentalmente, un hilo cuyo BTR es menor que un determinado umbral (low_btr) tiene una baja probabilidad de experimentar una degradación relevante causada por la contención de recursos compartidos. Al mismo tiempo, si el BTR combinado de los hilos que se ejecutan en un cluster de cores específico cae por debajo de un umbral específico ($high\_btr$) la degradación del rendimiento causada por la contención es habitualmente muy pequeña [46]. Tal y como se expone en [47, 46], los umbrales anteriormente citados pueden determinarse de forma simple a través de benchmarks sintéticos [47, 46]. De esta forma, se considera que la degradación del rendimiento de los hilos es 1 (no hay degradación) cuando se ejecutan en un core rápido en este tipo de situaciones de baja contención. En caso de que estos escenarios no se produzcan por sí solos causados por las asignaciones de hilos a cores realizadas por CAMPS para garantizar la justicia, el *core scheduler* pasará a un modo *non-work-conserving* (explicado en la sección 4.4), que se encargará de generar estos escenarios de baja contención de forma artificial.

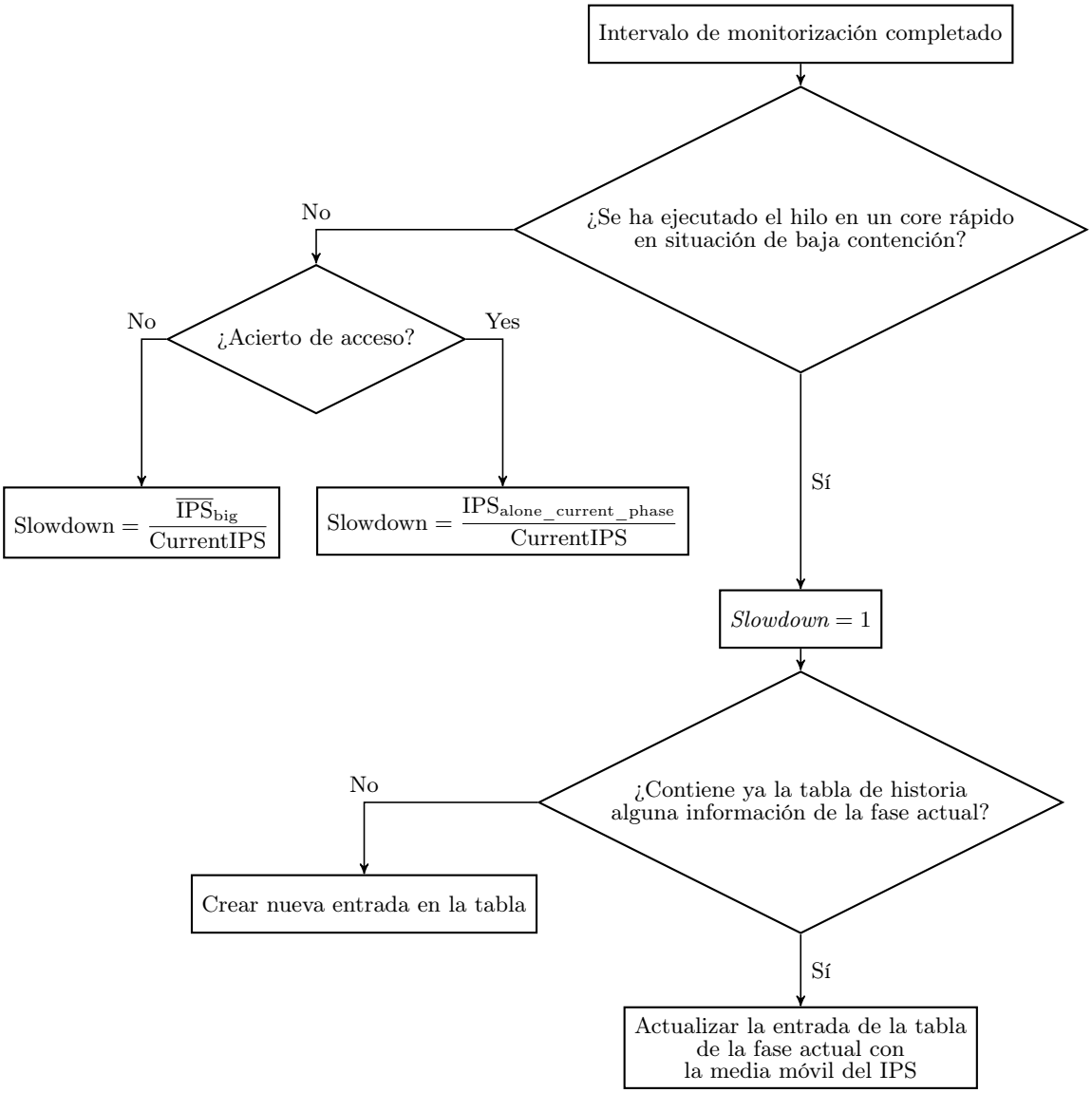


Figura 4.1: Mecanismo que CAMPS utiliza para estimar la degradación del rendimiento de un hilo con ayuda de la tabla de historia

Para indexar la tabla de historia de un hilo, necesaria para estimar la degradación del rendimiento y almacenar nuevas mediciones de IPS, se precisa que el *performance monitor* pueda determinar si existen datos almacenados en la tabla de historia para la fase de ejecución actual. Con este fin, se emplea una técnica de detección de fases similar a la empleada en investigaciones previas [22, 3]. La aproximación usada se basa en monitorizar periódicamente el porcentaje de instrucciones de cada tipo (int/FP, load, store y saltos) ejecutadas durante el último intervalo de monitorización. Este conjunto de valores se conoce como vector ITV (*instruction type vector*). En caso de que la distancia Manhattan de dos ITVs pertenecientes a diferentes muestras no supere un cierto umbral, se considera que ambas muestras pertenecen a una misma fase de ejecución. Para calcular la distancia Manhattan de dos vectores de dimensión n (X e Y) se utiliza la siguiente fórmula: $\sum_{i=1}^{n} |X_i - Y_i|$.

Lamentablemente, este mecanismo –cuya efectividad ha sido comprobada en simuladores [22, 3]– no se puede utilizar en las plataformas asimétricas reales empleadas en este trabajo, ya que sus unidades de monitorización del rendimiento (PMU) carecen de los eventos hardware requeridos. De hecho, aunque existiesen estos tipos de eventos no se podrían monitorizar de forma simultánea debido al limitado número de contadores hardware disponibles en las plataformas big.LITTLE de ARM utilizadas. Para solucionar estas limitaciones, se optó por recopilar el BTR e IPS de un hilo (información necesaria para la política de planificación) junto a dos métricas de control alternativas: el número de accesos a la caché de primer nivel (L1) por cada mil instrucciones, y el porcentaje de saltos retirados relativo al número total de instrucciones. De forma similar a los porcentajes de composición de instrucciones retiradas, los valores de las métricas mencionadas se mantienen estables para una fase concreta en situaciones con diferente grado de contención y, además, no difieren de forma sustancial entre tipos de core. De manera adicional, los valores de estas métricas de control experimentan unas variaciones significativas cuando una aplicación avanza a una nueva fase de ejecución que exhibe un comportamiento diferente en cuanto a intensidad en el uso de memoria y en cuanto a predicción de saltos. Estos dos aspectos juegan un papel clave en el rendimiento relativo de los dos tipos de core en las plataformas asimétricas [24, 38]. Por estas razones, consideramos a las métricas de control nombradas como una buena elección para indexar la tabla de historia de un hilo.

En la figura 4.1 se ilustra como el *performance monitor* estima la degradación del rendimiento de un hilo y mantiene su tabla de historia. La información de la tabla de historia es actualizada al final de un intervalo de monitorización en el que un hilo se ejecutó en un conjunto de cores rápidos con un bajo grado de contención. En caso de que la tabla no disponga de información de la fase actual,

el valor de IPS leído se almacena en una nueva entrada. Si ya existe información, se actualizará con la media móvil de los valores de IPS –recolectados en situaciones de baja contención– de esa fase. En ambos casos el planificador estima que la degradación del rendimiento de ese momento es 1 (no hay degradación). Por otra parte, si el hilo se ejecuta en un core lento, o en un core rápido en una situación de potencial contención, el *performance monitor* accede a la tabla de historia para estimar la degradación del rendimiento. En caso de que el valor de IPS para la fase actual se encuentre en la tabla (*phase hit*), la degradación del rendimiento se aproxima con la tasa del valor de IPS leído de la tabla ($\text{IPS}_{\text{alone\_current\_phase}}$) y el valor actual de IPS medido en el último intervalo de monitorización. En el caso de que no exista información de la fase actual en la tabla (*phase miss*), la degradación del rendimiento se aproxima con la tasa media de valores de IPS de todas las muestras almacenadas en la tabla para ese hilo ($\overline{IPS}_{big}$) y el valor de IPS medido actualmente.

Con el fin de determinar el tamaño de tabla más apropiado (máximo número de entradas), se analizaron diferentes trazas recolectadas con contadores hardware para 52 aplicaciones diferentes de la suite SPEC CPU. Pudimos observar que para cada aplicación se llega a un punto en el que incrementar más el máximo número de entradas de la tabla no proporciona ningún beneficio adicional. En concreto, para más del 90 % de las aplicaciones estudiadas, usar 20 entradas en la tabla de historia permite alcanzar la tasa de aciertos máxima observada (la tasa media de aciertos alcanzada por los benchmarks es 97.35 %). Para el resto de benchmarks ($< 10\,\%$) al usar 20 entradas en la tabla de fases garantiza al menos una tasa de aciertos del 82 %. Por último, cabe destcar que usar una tabla de historia de 20 entradas proporciona un buen equilibrio entre tasa de aciertos y coste de almacenamiento. En la implementación utilizada, una tabla de historia ocupa 940 bytes en plataformas de 64 bits (p.ej., placa Juno) y 760 bytes en plataformas de 32 bits (p.ej., Odroid XU4), lo cual supone menos de un 14 % del tamaño de la estructura de una tarea (*task_struct*).

4.3. Seguimiento del progreso e intercambio de hilos

El *core scheduler* del planificador mantiene contador de progreso para cada hilo conocido como **amp_progress**. El contador almacena el progreso realizado hasta el momento por el hilo relativo al progreso que habría realizado al ejecutarse en un core rápido todo el tiempo sin contención (sin compartir el cluster rápido con otros hilos). Cada vez que un hilo se ejecuta durante un *tick* de reloj, CAMPS incrementa `amp_progress` en $\Delta_{\texttt{amp\_progress}}$, que se calcula de la siguiente forma:

$$\Delta_{\texttt{amp\_progress}} = \frac{100 \cdot W_{\text{def}}}{CS \cdot W_t} \tag{4.1}$$

siendo W_t el peso del hilo, obtenido a partir de la prioridad de la aplicación definida por el usuario; W_{def} equivale al peso de las aplicaciones cuya prioridad es la predeterminada; y CS viene dado por la degradación del rendimiento actual estimada por el *performance monitor*. De forma similar a la aproximación utilizada por el planificador CFS de Linux para calcular el tiempo de CPU en sistemas simétricos [31], el mecanismo usado por CAMPS también tiene en cuenta la prioridad de los hilos en el cálculo del progreso.

A continuación se analiza un ejemplo que permite ilustrar la principal idea que está detrás de la definición de $\Delta_{\texttt{amp\_progress}}$. Supongamos que un programa secuencial con la prioridad por defecto ($W_{\text{t}} = W_{\text{def}}$) se está ejecutando en un sistema AMP, y su único hilo está asignado a un core rápido en una situación de baja contención. En este escenario, CS sería 1 (sin degradación), por lo que $\Delta_{\texttt{amp\_progress}}$ sería igual a 100. Esto indica que el hilo está haciendo actualmente el 100 % de su progreso máximo alcanzable, mientras se ejecuta en un core rápido sin contención. Consideremos ahora que el hilo se migra eventualmente a un core lento, donde experimenta una degradación del rendimiento relativa (CS) de 2.5; esta hipotética degradación del rendimiento se produce por la ejecución del hilo en un core mucho menos potente junto a la potencial degradación del rendimiento resultante al compartir los recursos con otros hilos. Bajo estas circunstancias, $\Delta_{\texttt{amp\_progress}}$ sería igual a 40; de esta forma, el hilo solo estaría realizando un 40 % de su rendimiento máximo alcanzable. Por lo tanto y de forma general, cuanto menor es la degradación del rendimiento (CS), más rápido se incrementa el contador `amp_progress` de un hilo.

Cuando un hilo nuevo entra en el sistema, el *core scheduler* lo asigna al core con menor carga de la plataforma de forma que se preserve un equilibrio en el reparto de trabajo. De forma similar a otras propuestas [27, 40], CAMPS prioriza los cores rápidos al asignar los hilos con el objetivo de maximizar el rendimiento global. Notablemente, el contador `amp_progress` de un nuevo hilo se inicializa al valor máximo de este contador observado entre los hilos del sistema en ese momento. Este valor inicial permite tratar de forma justa a aquellos hilos que han iniciado su ejecución en diferentes instantes de tiempo.

Además, cada hilo recién creado tiene también que atravesar una fase de calentamiento (10 intervalos de muestreo en la plataforma experimental utilizada) justo después de comenzar su ejecución. Las primeras dos muestras recolectadas durante el periodo de calentamiento son descartadas, con el objetivo de mitigar los fallos de predicción asociados a los numerosos fallos iniciales en la caché.

Es necesario remarcar que la aproximación utilizada por el *core scheduler* de CAMPS para garantizar la justicia a través de la monitorización y equilibrio del progreso tiene varios aspectos en común con el planificador ACFS [38]. En concreto, ambos planificadores mantienen contadores de progreso por hilo (actualizados cada *tick* de reloj). La diferencia fundamental reside en la forma de calcular en tiempo de ejecución la degradación del rendimiento actual (denominado como *CS* (*current slowdown*) en la ecuación 4.2), a diferencia de ACFS, CAMPS es consciente de la contención en recursos compartidos. De hecho, ACFS siempre asume que la degradación del rendimiento actual de un hilo siempre es 1 cuando se ejecuta en un core rápido, y utiliza el factor de ganancia de un hilo (aproximado mediante un modelo de predicción específico de plataforma) para estimar la degradación del rendimiento cuando el hilo se ejecuta en cores lentos.

De forma similar a ACFS, CAMPS también puede realizar intercambios de hilos entre cores de forma periódica para garantizar la justicia en el sistema. Esencialmente, los hilos asignados a cores rápidos por el planificador realizan habitualmente un progreso más rápido que los que se ejecutan en los cores lentos. Esta situación de injusticia puede detectarse de forma efectiva observando los contadores de progreso de los hilos. Ya que el factor de degradación del rendimiento actual o *CS* (*current slowdown*) es típicamente mayor cuando el hilo se ejecuta en un core lento, los contadores de progreso de los hilos que se ejecutan en cores lentos se incrementan a un ritmo menor que los que se ejecutan en cores rápidos.

Con el objetivo de equilibrar el progreso que hacen los diferentes hilos (justicia), CAMPS realiza migraciones de hilos siguiendo unos criterios parecidos a los de ACFS. Específicamente, un hilo que se ejecuta en un core rápido será intercambiado con otro hilo que se ejecuta en un core lento solamente cuando la diferencia entre sus contadores de progreso sobrepasa un cierto umbral, conocido como `amp_threshold`. Las instrucciones específicas seguidas para seleccionar el valor más apropiado de este umbral para cada plataforma se proporcionan en [38].

Se ha comprobado que basarse únicamente en los contadores de progreso de los hilos (de forma similar a ACFS) es ineficaz en el caso de que aplicaciones agresoras y programas sensibles a la contención se asignen simultáneamente al conjunto de cores rápidos. Tal y como se explica en el capítulo 2, esta asignación puede provocar que las aplicaciones sensibles a la contención experimenten una degradación del rendimiento excepcionalmente alta, lo que puede reducir los beneficios obtenidos al usar un core rápido hasta el punto de que sean inexistentes. Para mitigar este aspecto, CAMPS utiliza la métrica Bus Transfer Rate (BTR) utilizada en [46] para identificar situaciones de posible contención, y promueve aquellas migraciones que ayuden a evitar la contención en el cluster de cores rápidos. El algoritmo 1 ilustra el mecanismo usado por el *core scheduler* de CAMPS para decidir los intercambios de hilos. Fundamentalmente, el planificador siempre selecciona el hilo con el mayor

Algorithm 1: Selección de los candidatos de intercambio en CAMPS

Input: T_B es el hilo listo para ejecutarse con el contador de progreso más alto asignado a un core rápido, S es el conjunto de hilos listos para ejecutarse (T_S, i) asignados a cores lentos que constituyen un potencial candidato de intercambio por T_B (i.e. `amp_progress`(T_B)−`amp_progress`(T_S, i)≥`amp_thresh`). Nótese que $S \neq \varnothing$, y los hilos en S están ordenados ascendentemente por sus contadores `amp_progress`.

```
min_btr ← ∞;
T_min-BTR ← NIL;
intercambio_realizado ← false;
do
    T_S, i ← Obtener el primer hilo en S ;
    if Intercambiar T_B y T_S, i lleva a una situación de baja contención en
     el conjunto de cores rápidos ||
     (amp_progress(T_B)−amp_progress(T_S, i) ≥ 2∗amp_threshold) then
        Intercambiar T_B y T_S;
        intercambio_realizado ← true;
    else
        Eliminar T_S, i de S;
        if BTR(T_S, i) < min_btr then
            min_btr ← BTR(T_S, i) ;
            T_min-BTR ← T_S, i ;
        end
    end
while !intercambio_realizado && S ≠ ∅;
if !intercambio_realizado then
    Intercambiar T_B y T_min-BTR;
end
```

contador `amp_progress` asignado a un core rápido –denotado como T_B– como candidato de intercambio para ser migrado a un core lento. Al seleccionar otro hilo a intercambiar, CAMPS considera en primer lugar aquellos hilos de los cores lentos con un menor valor del contador `amp_progress` (hilos cuyo progreso se está quedando atrás respecto al resto). En el caso de que sea posible realizar un intercambio que disminuya el grado de contención en el cluster de cores rápidos, CAMPS se encargará de realizarlo. Si este no es el caso, el planificador seleccionará el hilo a intercambiar con el menor BTR entre los elegibles para realizar el intercambio (aquellos cuyos contadores de progreso sean más bajos). Tal y como se expone en [46], cuando se reduce el BTR agregado en un cluster de cores se ayuda a reducir la contención de recursos compartidos. Cabe destacar que el planificador impone como candidato de intercambio aquellos hilos cuya ejecución se está ralentizando en mayor medida, específicamente, si la diferencia entre el contador de progreso de

T_B y el contador de progreso del hilo es mayor que 2∗`amp_threshold`. Esto hace posible que los hilos agresores, con un alto BTR, puedan tener la oportunidad en algún momento de ejecutarse en cores rápidos en aquellas situaciones en las que la carga de trabajo incorpora múltiples aplicaciones intensivas en memoria.

4.4. Modo non-work conserving

Tal y como se expuso en la sección 4.2, el planificador CAMPS se basa en la capacidad de medir el rendimiento de un hilo mientras se ejecuta en cores rápidos en una situación de baja contención. El *performance monitor* es capaz de reconocer estas situaciones de baja contención y actualizar la tabla de historia con nuevas lecturas de IPS. Estas muestras serán usadas para obtener una predicción de la degradación del rendimiento de un hilo mientras se ejecuta en diferentes tipos de core y con distintos grados de contención. Como se explica en [46], los escenarios de baja contención se producen de forma natural para aquellos hilos que no son sensibles a la contención debido a que tienen un BTR bajo (p.ej., su *conjunto de trabajo* cabe en la caché de primer nivel) o cuando un hilo que es potencialmente sensible (alto BTR) se ejecuta simultáneamente con otros hilos cuyo BTR agregado no supera un cierto umbral.

Desafortunadamente, si el número de hilos en la carga de trabajo que hacen un uso intensivo de memoria es alto, las situaciones de baja contención podrían no suceder de forma tan habitual como es necesaria para un correcto funcionamiento del planificador. Bajo estas circunstancias, el *core scheduler* puede activar el modo *non-work-conserving* (NWC), en el cual las situaciones de baja contención se generan artificialmente. La lógica que hay detrás de la activación de este modo se explica a continuación. Siempre que un hilo completa k intervalos de monitorización (siendo k un parámetro modificable), el planificador obtiene tanto la tasa de aciertos del hilo como el número de muestras de IPS agregadas a la tabla durante ese intervalo. En caso de que la tasa de aciertos caiga por debajo del 80 %, y no se haya añadido una sola muestra de IPS durante el intervalo de tiempo, CAMPS activa el modo NWC. El hilo que causó la transición a este modo se denomina *hilo NWC*.

Mientras el planificador se encuentre en el modo NWC no se realizarán los intercambios de hilos que habitualmente se hacían para garantizar la justicia. Durante este modo especial, el principal objetivo es recolectar el mayor número posible de muestras de IPS en cores rápidos del hilo NWC que no estén afectadas por la contención. Para ello, si este hilo no se encontraba ya ejecutándose en un core rápido, será intercambiado con un hilo intensivo en memoria (alto BTR) con el objetivo de reducir al máximo posible la contención en el cluster de cores rápidos. Una vez el hilo NWC está asignado a un core rápido, el planificador comenzará a recopilar

muestras de IPS en cores rápidos con un bajo grado de contención. Si todavía no se ha producido un escenario de baja contención tras el intercambio, el planificador desactivará temporalmente (durante un intervalo muy corto de tiempo) tantos cores rápidos como sea necesario. En realidad, para generar esta situación solo se necesitará deshabilitar los cores en los que se estén ejecutando hilos intensivos en memoria. Cabe destacar que durante este escenario de baja contención, otros hilos que se ejecuten en un core rápido (además del hilo NWC) pueden tener la oportunidad añadir muestras de IPS a su tabla de historia.

El planificador realizará la transición al modo normal de funcionamiento cuando (1) la tasa de aciertos del hilo NWC se encuentre por encima del 80 %, que puede producirse rápidamente tras haber insertado algunas muestras de IPS en la tabla de historia; o (2) cuando el hilo NWC se bloquea o termina. Notablemente, cuando el modo NWC está activo, el planificador continua aumentando los contadores de progreso de los hilos activos. De esta forma, se hace posible que aquellos hilos que no se beneficiaron por la activación del modo NWC, aquellos asignados a cores rápidos que fueron desactivados temporalmente, sean compensados más adelante. De forma similar, para evitar que hilos específicos provoquen la activación del modo NWC de forma reiterada se tienen en cuenta los contadores de progreso para controlar las transiciones a este modo; aquellos hilos que realicen un progreso mucho mayor que el resto no pueden convertirse en hilos NWC.

Por último, es necesario remarcar que varias estrategias de planificación para multicores simétricos [12, 46] también explotan las desactivaciones de cores para crear escenarios de baja contención de forma artificial. Sin embargo, estas estrategias realizan las operaciones de desactivación a nivel de usuario en lugar de a nivel del kernel (como se hace en este trabajo). En concreto, modificando la operación *pick_ next_ task* de la clase de planificación de CAMPS. Los cores rápidos se desactivan temporalmente seleccionando la tarea *idle* para que se ejecute de forma obligatoria en el core correspondiente y, además, se modifica la afinidad del hilo previamente asignado para evitar que sea migrado a otro core por el equilibrador de carga. Observamos que al usar un período de desactivación de cores corto, de 100 ms en nuestra plataforma experimental (2 intervalos de monitorización), se permite que el planificador tenga un control de grano fino durante el modo NWC. Esencialmente, esto permite que CAMPS ajuste mejor el número de operaciones de desactivación necesarias demandadas por el hilo NWC actual.

4.5. Soporte especial para aplicaciones multihilo

Con el fin de proporcionar un mejor soporte para aplicaciones multihio, el planificador CAMPS emplea dos mecanismos especiales: *notificaciones de espera activa* y *tablas de historia por aplicación.*

Las *notificaciones de espera activa* permiten que el planificador sea consciente de aquellas situaciones en las que los hilos de un programa multihilo hacen una espera activa (*spin*) en lugar de bloquearse mientras esperan para que otros hilos alcancen puntos de sincronización concretos (p.ej., barreras). La espera activa puede reducir de forma sustancial el número de cambios de contexto realizados por el planificador del sistema operativo [20], y también puede reducir el número de migraciones en multicores asimétricos [40]. No obstante, los hilos que hacen una espera activa deben de ser gestionados correctamente por el planificador. El principal problema se produce porque al realizar esta espera activa los hilos pueden alcanzar unos valores altos de IPS a pesar de no realizar un trabajo útil[1]. Por tanto, si CAMPS utiliza estos valores de IPS, la tabla de historia se llenaría con muestras que no representan el rendimiento habitual del hilo, lo que causaría futuros errores en las predicciones de la degradación del rendimiento y del factor de ganancia.

En este trabajo se utiliza una variante de la técnica de *notificaciones de espera activa* propuesta en trabajo previo [40]. En la implementación utilizada, se mantiene una región de memoria compartida entre cada hilo de la aplicación y el sistema operativo. Cuando un hilo inicia su espera activa, activa un flag en la región de memoria compartida, que más tarde será desactivado cuando reanude su ejecución regular. Se ha optado por esta alternativa en lugar de utilizar llamadas al sistema (como en [40]) para evitar una sobrecarga adicional asociada a las llamadas al sistema. En el caso de que un hilo se encuentre realizando una espera activa, el *performance monitor* descarta las muestras de IPS asociadas, y estima siempre como 1 la degradación del rendimiento y el factor de ganancia relativos (al no hacer trabajo útil). De igual modo, el *core scheduler* de CAMPS deja de migrar a cores rápidos aquellos hilos que se encuentren realizando una espera activa. Nótese que, realizar notificaciones de espera activa desde espacio de usuario no requiere modificar las aplicaciones siempre cuando éstas usen las primitivas de sincronización proporcionadas por la librería de hilos o el *runtime system*. Como prueba de concepto, se implementó este mecanismo en el *runtime system* OpenMP de GCC. Se instrumentó el código de las primitivas de sincronización incluidas en la librería dinámica *libgomp*. Esta aproximación puede utilizarse para implementar notificaciones de espera activa en la librería *POSIX threads*.

En muchas aplicaciones multihilo, los diferentes hilos realizan las mismas operaciones pero con diferentes datos. En este escenario, se pueden reutilizar las muestras de IPS almacenadas en la tabla de historia de un hilo para mejorar la predicción de la degradación del rendimiento del resto de hilos de la aplicación. Para ello,

[1]Cabe destacar que las mejores prácticas al implementar *spinlocks* recomiendan que un hilo itere sobre una variable local [2], lo que produce un mayor IPC, debido a la utilización eficiente del pipeline de la CPU.

se mantienen dos niveles de tabla de historia para las aplicaciones multihilo: la tabla por cada hilo (L1) presentada en la sección 4.2, y una tabla de historia por aplicación (L2). Fundamentalmente, cuando un hilo crea una entrada nueva para su tabla de historia, inserta también esta entrada en la tabla a nivel de aplicación. De este modo, los otros hilos de la aplicación que experimenten un fallo en el acceso a la L1, pueden potencialmente recuperar la información para su fase actual accediendo a la tabla L2. Si se produce un acierto accediendo a la L2, la entrada se copia a la tabla privada del hilo (L1). Así se consiguen evitar futuros accesos a la misma entrada de la L2, por lo que se reduce la potencial contención que puede generarse si múltiples hilos intentan acceder esta tabla compartida (protegida con un cerrojo).

A pesar de que usar dos niveles de tabla de historia es especialmente recomendable para aquellas aplicaciones en las que todos los hilos ejecutan el mismo código pero con diferentes datos, esta estrategia podría extenderse fácilmente a otros tipos de aplicaciones multihilo (como los que siguen el paradigma del *pipeline*) donde unos pocos hilos realizan una tarea específica de forma cooperativa, mientras que otros se encargan de otra tarea completamente diferente. En ese caso, la tabla de historia L2 podría ser compartida por aquellos que ejecutan una tarea similar, que pueden identificarse por la función que ejecutan.

Merece la pena destacar que CAMPS tiene en cuenta el número de hilos activos de una aplicación multihilo –un indicador para el grado de paralelismo a nivel de hilo– para aproximar de una forma más precisa el factor de ganancia a nivel de aplicación y la degradación del rendimiento cuando se monitoriza el progreso. En investigaciones previas se ha demostrado que al tomar decisiones de planificación basándose en el factor de ganancia o la degradación del rendimiento de cada hilo por separado puede provocar una degradación significativa del rendimiento global cuando varias aplicaciones multihilo forman parte de la carga de trabajo [39, 38]. Esto surge del hecho de que el factor de ganancia (SF) no aproxima el beneficio global que una aplicación multihilo deriva de utilizar todos los cores disponibles en la plataforma asimétrica (rápidos y lentos) frente a usar solo los cores lentos [4, 16]. Básicamente, cuando el número de hilos de la aplicación excede la cantidad de cores rápidos disponibles algunos hilos pueden ser relegados a los cores lentos. Finalmente, si los hilos se sincronizan entre sí de forma frecuente, asignar los hilos a diferentes tipos de core simultáneamente puede provocar un progreso desigual y un uso ineficiente de los cores [39, 45, 38]. Esto sucede en muchas aplicaciones paralelas usadas en nuestros experimentos, que provienen del entorno de la computación de altas prestaciones. Para evitar este problema, aplicamos un factor de escalado de

los factores de ganancia y de degradación del rendimiento del hilo en proporción al número de cores rápidos disponibles, siguiendo la aproximación de [39, 38]. Estos valores escalados (en lugar de los valores por hilo) se utilizan en las ecuaciones 4.2 y 4.3 para los hilos que pertenecen a aplicaciones paralelas.

4.6. Ajuste del compromiso rendimiento-justicia

Como ya se ha mencionado, el mecanismo de seguimiento del progreso utilizado por CAMPS está inspirado en el planificador ACFS. Una de las principales ventajas al tener en cuenta las prioridades de las aplicaciones cuando se registra el progreso como ACFS, es que ya se ha demostrado que esta aproximación resulta efectiva a la hora de considerar las prioridades establecidas por el usuario en hardware asimétrico real [38]. Otro aspecto de gran utilidad del mecanismo de seguimiento del progreso de ACFS es que este puede extenderse con un parámetro configurable (referido como *unfairness factor*), que permite al planificador incrementar el rendimiento global del sistema asimétrico en escenarios en los requisitos de justicia no sean tan estrictos. Al mismo tiempo, al usar valores altos de este parámetro el planificador optimizará el rendimiento global en lugar de la justicia.

Para permitir al administrador del sistema degradar la justicia a costa de mejorar el rendimiento global en función de las necesidades de cada momento, procedimos a incorporar a la implementación de CAMPS el parámetro *unfairness_factor* (UF). Para comprender completamente las diferencias entre la implementación de este parámetro en CAMPS y ACFS, presentamos en primer lugar los aspectos comunes. Para conseguir mejorar el rendimiento global del sistema asimétrico, el planificador debe otorgar un mayor tiempo de uso de los cores rápidos a aquellas aplicaciones que deriven un mayor factor de ganancia o *speedup* de usar este tipo de cores con respecto a los lentos. Con esta finalidad, se empleó una estrategia de prioridad dinámica, en el que la prioridad actual de un hilo depende de su peso estático (establecido por el usuario) y del factor de ganancia relativo de la aplicación (S_{BS}).

$$\Delta_{\texttt{amp\_progress}} = \frac{100 \cdot W_{\text{def}}}{CS \cdot W_t} \tag{4.2}$$

Para implementar esta estrategia se reemplaza el peso estático (W_t) de la ecuación 4.2 (o la correspondiente ecuación de ACFS [38]) con el peso dinámico (DW_t), que se define de la siguiente forma:

$$DW_t = W_t \cdot \left(1 + \frac{(\texttt{UF} - 1) \cdot (S_{BS} - S_{min})}{S_{max} - S_{min}}\right) \tag{4.3}$$

donde S_{max} y S_{min} son los factores de ganancia máximos y mínimos observados entre las aplicaciones de la carga de trabajo.

Cuando el parámetro UF se establece a su valor por defecto y mínimo (1.0), entonces tenemos que $DW_t = W_t$, por lo que el planificador se comporta de forma estándar intentando optimizar la justicia. Con valores de $UF > 1$, el planificador mejora el rendimiento global a costa de degradar la justicia gradualmente. Para ello, sustituye el peso estático (W_t) por su versión dinámica (DW_t).

El planificador debe estar equipado con un mecanismo para predecir en tiempo de ejecución el factor de ganancia relativo de cada hilo para ser capaz de calcular su peso dinámico (DW_t). Tal y como se describe en [38], ACFS emplea modelos de predicción específicos de plataforma para aproximar el factor de ganancia de un hilo mediante contadores hardware. Esta aproximación podría utilizarse en CAMPS también, pero limitaría la portabilidad del planificador. Esencialmente, al basarse en este tipo de modelos de estimación se necesita monitorizar un conjunto específico de eventos de rendimiento que dependen en gran medida del tipo de asimetría de la plataforma [39], y que están típicamente limitados al modelo de procesador del sistema AMP [24, 38].

Para garantizar que el planificador sea portable entre diferentes plataformas asimétricas, se utilizan los valores de IPS recolectados en ambos tipos de core para aproximar el factor de ganancia. Nótese que para mejorar la predicción, se emplea la información almacenada en una tabla de historia de cada hilo que contiene valores de IPS de la misma fase de ejecución siempre que estén disponibles. Para ello, se añadió un nuevo campo en cada entrada de la tabla de historia, referido como $IPS_{max,small}$, que almacena los valores máximos de IPS observados para la fase del programa en cuestión cuando se ejecutó en un core lento. Debido a que el rendimiento de un hilo es habitualmente menor en situaciones de contención, $IPS_{max,small}$ constituye una cota inferior del IPS que se observaría cuando el hilo se ejecutase durante esa fase solo en el core lento. Al final de cada intervalo de monitorización el *performance monitor* accede a la tabla de historia y la actualiza si es necesario. En el caso de que se produzca un acierto en la tabla de historia, el SF del hilo se aproxima como $\text{IPS}_{\text{alone\_current\_phase}}/IPS_{max,small}$. En caso contrario, el factor de ganancia del hilo se estima como $\overline{IPS}_{big}/\overline{IPS}_{small}$, donde $\overline{IPS}_{small}$ es la media de valores de IPS recogidos para un hilo en un core lento.

Capítulo 5

Evaluación experimental

En este capítulo se comienza realizando una comparativa de la efectividad de CAMPS con respecto a otros planificadores conscientes de la asimetría propuestos previamente [7, 24, 45, 38], que se centran principalmente, igual que CAMPS, en la planificación de aplicaciones de larga duración intensivas en cómputo. Todos estos planificadores (evaluados en la sección 5.1) se han implementado en este trabajo como una clase de planificación independiente en la versión 3.10 del kernel Linux. En la sección 5.2, se ilustra la efectividad que tiene el mecanismo incluido en CAMPS para mejorar el rendimiento global a costa de degradar la justicia.

5.1. CAMPS contra otros planificadores conscientes de la asimetría

Para evaluar la efectividad de CAMPS, se hace una comparación con otros tres planificadores conscientes de la justicia previamente propuestos para multicores asimétricos: ACFS [38], Equal-Progress [45] y un planificador que reparte de forma equitativa los ciclos de core rápido entre aplicaciones utilizando una estrategia round-robin (RR) [7]. Por ofrecer una perspectiva más amplia, también se experimentó con un planificador cuyo objetivo es optimizar el rendimiento global asignando de forma prioritaria a los cores rápidos aquellas aplicaciones con un mayor factor de ganancia relativo [24, 39]; nos referiremos a este planificador como *HSP* (High SPeedup).

Exceptuando a Round-Robin, todos los algoritmos de planificación evaluados requieren para su funcionamiento obtener información de contadores hardware (*Performance Monitoring Counters* o PMCs). Para obtener la información de estos contadores, se ha utilizado como interfaz a nivel del kernel la herramienta de código abierto PMCTrack [37]. Los planificadores HSP y ACFS aproximan el SF

de los hilos recogiendo varios eventos hardware de los PMCs y proporcionando esta información a un modelo de predicción específico de plataforma. En [41], se puede encontrar la información necesaria para construir los modelos de estimación de SF y determinar los eventos asociados a cada una de nuestras plataformas experimentales. Por el contrario, el planificador Equal-Progress [45] se basa en el uso de Performance Impact Estimation (PIE) [44] o bien en muestreo de IPC [7] para determinar el factor de ganancia en tiempo de ejecución. Debido a que las extensiones hardware especiales que PIE requiere no están disponibles en las plataformas AMP comerciales, hemos evaluado la variante de Equal-Progress basada en historia que utiliza muestreo de IPC.

En la implementación de todos los planificadores, la información de los contadores hardware se lee para cada hilo en intervalos de 50 ms; este período de muestreo permite detectar fases de grano grueso de un programa desde el sistema operativo, y también filtrar variaciones bruscas de las métricas de rendimiento que se producen cuando se usan periodos más cortos (debido a rápidas oscilaciones de algunas métricas). Al mismo tiempo, se observó que la sobrecarga asociada con el procesamiento relacionado a los contadores hardware es despreciable a esta frecuencia de muestreo. Para experimentar una sobrecarga apreciable (alrededor del 1 %) en las plataformas exploradas, el período de muestreo tiene que reducirse a 5 ms.

En los experimentos realizados, se han utilizado las configuraciones asimétricas 2B-4S y 4B-4S (basadas en procesadores big.LITTLE de ARM) que se presentan en la sección 2.2.1. Esta evaluación experimental se centra en cargas de trabajo formadas por aplicaciones intensivas en CPU de larga duración de diversas suites de benchmarks (SPEC CPU, PARSEC, Minebench y NAS Parallel). También se experimentó con `FFTW3D`, un programa que realiza la transformada rápida de Fourier. Todos los programas se compilaron con GCC (nivel de optimización `-O3`) y utilizando las opciones `-mtune=cortex-a15.cortex-a7` (en 4B-4S) y `-mtune=cortex-a57.cortex-a53` (en 2B-4S) para aplicar optimizaciones comunes para los cores rápidos y lentos de las plataformas AMP. En todos los experimentos, el número de hilos total de la carga de trabajo coincide con el número total de cores de la plataforma, de la misma forma en la que se ha realizado en trabajo previo sobre sistemas multicore asimétricos [24, 39, 45]. Al lanzar cada carga de trabajo de nuestros experimentos garantizamos que todas las aplicaciones de la carga de trabajo se lancen simultáneamente y que cuando una termine, se reinicie de forma reiterada hasta que la aplicación de mayor duración se haya ejecutado tres veces. Acabada la ejecución de la carga se mide el grado de injusticia (*un-*

Tabla 5.1: Cargas de trabajo multiprogramadas para la configuración asimétrica 2B-4S.

Nombre	Aplicaciones
W1	GemsFDTD,equake,soplex,milc,ammp,bzip2
W2	galgel,soplex,hmmer,lbm,fma3d,bzip2
W3	galgel,equake,gamess,lbm,bzip2,astar
W4	twolf,bwaves,equake,soplex,astar,gobmk
W5	GemsFDTD,bwaves,equake,povray,fma3d,astar
W6	bwaves,equake,gamess,lbm,fma3d,bzip2
W7	GemsFDTD,applu,perlbmk,sixtrack,astar,gzip
W8	bwaves,perlbmk,povray,fma3d,astar,gzip
W9	galgel,perlbmk,sixtrack,mgrid,astar,libquantum
W10	GemsFDTD,vortex,perlbmk,fma3d,astar,gzip
W11	bzip2,equake,hmmer,vortex,crafty,astar
W12	gamess,hmmer,soplex,art,astar,gzip
W13	GemsFDTD,bwaves,gamess,hmmer,crafty,astar
W14	bzip2,bwaves,hmmer,lucas,gobmk,gzip
W15	soplex,art,vortex,lbm,fma3d,gobmk
W16	galgel,equake,hmmer,lbm,fma3d,h264ref
W17	bwaves,equake,gamess,povray,astar,libquantum
W18	GemsFDTD,galgel,gamess,hmmer,astar,libquantum
W19	swim,mcf,perlbench,h264ref,gobmk,gzip
W20	galgel,equake,hmmer,povray,mgrid,gobmk
W21	galgel,equake,hmmer,bzip2,perlbench,h264ref
W22	galgel,equake,gamess,hmmer,sixtrack,povray
W23	gamess,art,bzip2,gobmk,sixtrack,vortex
W24	galgel,gamess,hmmer,povray,perlbench,gobmk

fairness) y el rendimiento global (*throughput*) para el planificador con el que se haya experimentado, para lo que se calcula la media geométrica de los tiempos de ejecución de cada programa. Para cuantificar el rendimiento global se empleó la métrica *Aggregate Speedup* (ASP), de igual modo que en trabajos previos [38, 41].

Con el fin de analizar la efectividad de los diferentes planificadores se elaboraron dos conjuntos de cargas de trabajo, mostradas respectivamente en las tablas 5.1 y 5.3. En la primera, cada combinación de programas consta de seis aplicaciones monohilo que se ejecutan en la configuración asimétrica 2B-4S. El segundo conjunto, que se ejecutó en la configuración 4B-4S, se compone de mezclas de aplicaciones paralelas y secuenciales.

5.1.1. Cargas de trabajo en la configuración 2B-4S

En primer lugar, se analizarán los resultados del primer conjunto de cargas de trabajo, mostrados en la figura 5.1. Los valores de injusticia (unfairness) y rendimiento global (*ASP*) que se presentan en las gráficas están normalizados con respecto a los resultados del planificador HSP. Al construir las cargas de trabajo (tabla 5.1), dividimos las aplicaciones SPEC CPU en dos grupos: programas que

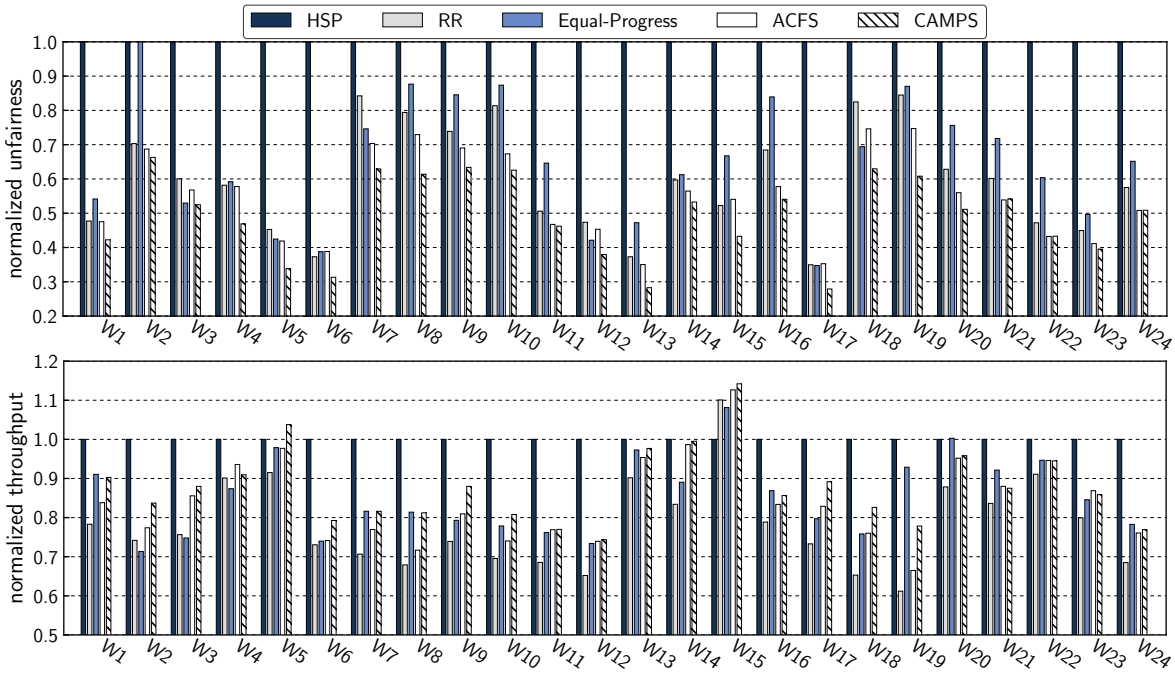


Figura 5.1: Valores de injusticia (arriba) y rendimiento global (abajo) para las cargas de trabajo de la tabla 5.1 ejecutándose en la configuración 2B-4S bajo los diferentes algoritmos de planificación

que exhiben fases mayoritariamente intensivas en CPU, cuyo rendimiento no sufre significativamente en situaciones de contención; y programas intensivos en memoria, los cuales están sujetos a una importante degradación del rendimiento causada por la contención o ejercen una presión significativa en los recursos compartidos. Teniendo en cuenta los diferentes tipos de aplicación, realizamos mezclas usando 29 benchmarks diferentes de SPEC CPU que exhiben un amplio espectro de factores de ganancia.

En la tabla 5.1 se muestran las mezclas de programas ordenadas en orden descendente por el número de aplicaciones intensivas en memoria que se incluyen en la carga de trabajo. Por ejemplo, la primera carga de trabajo está compuesta por seis programas que hacen un uso intensivo de memoria, mientras que la última está formada por cinco aplicaciones intensivas en CPU y una aplicación sensible a la contención (`galgel`).

Los resultados demuestran que al optimizar una métrica se produce una degradación sustancial de las otras. Este comportamiento es consistente con las conclusiones obtenidas en trabajos anteriores [38, 41], que ilustran que la justicia y el rendimiento global son objetivos de optimización contrapuestos en sistemas multicore asimétricos. Como se aprecia en los resultados, el planificador HSP (que optimiza el rendimiento global) alcanza las mejores cifras de ASP para la mayoría de cargas de trabajo, con la contrapartida de obtener generalmente los peores valores de unfairness (cuanto mayor peor). De igual modo, el resto de planificadores (orientados a mejorar la justicia), consiguen una mejora sustancial del *unfairness* con respecto a HSP (CAMPS mejora hasta un 72 % en la carga W17), a costa de una degradación del rendimiento global que puede llegar a ser muy significativa (Round-Robin llega a empeorar hasta un 38 % en la carga W19).

Los resultados de los planificadores ACFS, RR y CAMPS muestran una clara tendencia para la mayoría de de cargas de trabajo. ACFS ofrece un mayor rendimiento global y mayores reducciones de la injusticia (unfairness) que Round-Robin. Este es el comportamiento esperado, ya que ACFS, a diferencia de Round-Robin, tiene en cuenta el factor de ganancia de cada aplicación cuando reparte el tiempo de ejecución en cores rápidos. Aunque ACFS en general muestra un mayor rendimiento global, como este planificador no tiene en cuenta la contención en recursos compartidos al tomar decisiones de planificación, éste ofrece niveles similares de injusticia a los de Round-Robin para algunas cargas de trabajo (p.ej W4-W6, W15 or W17). En contraposición, el planificador CAMPS llega a reducir la injusticia todavía más: hasta un 11 % relativo a ACFS (W17) y hasta un 28 % con respecto a Round-Robin (W19). De igual modo, CAMPS puede llegar a mejorar el rendimiento global hasta en un 17 % con respecto a ACFS (W19). Notablemente, en aquellas cargas de trabajo con bajos niveles de contención que tienen una menor cantidad de aplicaciones intensivas en memoria (W20-W24), se puede apreciar que ACFS y

Tabla 5.2: Reducción media de la injusticia (unfairness) e incremento del rendimiento global (throughput) obtenidos por CAMPS con respecto al resto de algoritmos de planificación en la placa Juno de ARM.

CAMPS vs. otros	**Reducción de injusticia**	**Incremento del rendimiento**
HSP	50.96 %	-12.24 %
RR	17.08 %	13.19 %
Equal-Progress	23.64 %	3.31 %
ACFS	10.71 %	4.48 %

Tabla 5.3: Cargas de trabajo multiprogramadas para la configuración asimétrica 4B-4S.

Nombre	Aplicaciones
M1	art,galgel,libquantum,sixtrack,gamess,hmmer,soplex,gzip
M2	galgel,libquantum,hmmer,mcf,mgrid,crafty,parser,gzip
M3	mcf,lucas, galgel, soplex,h264ref,povray,perlbmk,gobmk
M4	galgel,mcf,h264ref,povray,perlbmk,crafty,gobmk,astar
M5	gamess,hmmer,mcf,mgrid,lucas,applu,namd,gobmk
M6	art,galgel,gamess,hmmer,mgrid,lucas,h264ref,astar
M7	hmmer,mcf,mgrid,lucas,soplex,applu,h264ref,gzip
M8	mgrid,lucas,soplex,fma3d,applu,ammp,h264ref,astar
M9	art,libquantum,sixtrack,h264ref,semphy(4)
M10	gamess,applu,povray,crafty,swaptions(4)
M11	soplex,povray,namd,gobmk,semphy(4)
M12	fma3d,h264ref,povray,equake,kmeans(4)
M13	FFTW3D(4),kmeans(4)
M14	semphy(4),EP(4)
M15	blackscholes(4),EP(4)
M16	blackscholes(4),kmeans(4)

CAMPS consiguen cifras muy similares para ambas métricas de optimización. Esta observación sugiere que la estrategia de planificación propuesta en este trabajo es también adecuada para escenarios de baja contención, ya que proporciona cifras similares de rendimiento global y justicia que ACFS, que es el planificador que ofrece mejores resultados en estas circunstancias y que está considerado el planificador de referencia en cuanto a justicia para sistemas multicore asimétricos [38]. Teniendo en cuenta todo esto, como queda reflejado en la tabla 5.2 el algoritmo de planificación CAMPS consigue una reducción media del 10.7 % en la injusticia con respecto a ACFS mientras mejora el rendimiento global en un 4.48 %.

Ahora nos centramos en el análisis de los resultados del algoritmo de planificación Equal-Progress, que también intenta optimizar la justicia en multicores asimétricos. Claramente, Equal-Progress no es capaz de obtener cifras inferiores de injusticia (*unfairness*) para la mayor parte de cargas de trabajo. Además, se pueden observar divergencias significativas en los resultados de las distintas cargas de trabajo; en algunas cargas (p.ej., W3, W4, W12, W17 y W18) Equal-Progress

obtiene unas cifras de justicia y rendimiento global cercanas a las de CAMPS y ACFS, mientras que para otras (p.ej., W2, W8-W10, W14-W16) muestra un comportamiento mucho más injusto. Tal y como se discute en trabajos anteriores [38], en los que se observaron divergencias similares, esta disparidad de comportamiento surge de dos factores principales: (1) la falta de precisión del mecanismo que utiliza Equal-Progress para realizar el seguimiento del progreso de un hilo en el sistema asimétrico, y (2) el hecho de que requiere emplear muestreo de IPC [45] para determinar los factores de ganancia o speedup factor (SF) en tiempo de ejecución. Se ha demostrado que esta técnica de muestreo de IPC ha resultado imprecisa para aproximar el factor de ganancia de muchas aplicaciones, ya que las muestras de IPC recogidas en cada tipo de core pueden pertenecer a diferentes fases de ejecución [42]. En nuestros experimentos hemos observado que las inexactitudes al predecir el factor de ganancia son más frecuentes en presencia contención, ya que el IPC de una aplicación puede sufrir bruscas oscilaciones (incluso dentro de la misma fase del programa) en función del grado de contención que esté experimentando el hilo en cada momento. A pesar de que CAMPS también mide el IPC de un hilo para determinar el factor de degradación del rendimiento, los valores de referencia tomados para aproximar el rendimiento de su ejecución en aislamiento (IPS_{alone}) son aquellos almacenados en la tabla de historia del hilo cuando este ejecutó la misma fase en un core rápido sin contención. Esto hace posible que CAMPS no se vea afectado por el problema mencionado de Equal-Progress, y le permite obtener los mejores valores de justicia (menores cifras de *unfairness*) de todas las estrategias estudiadas. Tal y como se muestra en la tabla 5.2, CAMPS disminuye la injusticia hasta un 23.6 % en media relativo a Equal-Progress.

Los resultados también revelan que para algunas cargas de trabajo CAMPS y ACFS obtienen unas cifras de rendimiento global similares a las de HSP (en un rango del 3 %). De forma general, se ha observado que la degradación del rendimiento global generada por los planificadores conscientes de la justicia es significativamente menor para aquellas cargas de trabajo en las que el número de aplicaciones que tienen un alto factor de ganancia supera al número de cores rápidos disponibles (dos). En estas circunstancias (p.ej., W1, W3, W13, o W17), CAMPS y ACFS ofrecen la misma oportunidad a todas estas aplicaciones de los cores rápidos de forma proporcional (mediante migraciones periódicas), mientras que HSP simplemente asigna dos de las aplicaciones de alto SF a los cores rápidos disponibles durante un largo periodo de tiempo. Este comportamiento provoca que ACFS y CAMPS reduzcan la injusticia en mayor medida que HSP (p.ej, W13 y W5), mientras que causan una baja degradación del rendimiento global. En resumen, el intercambio periódico de hilos permite reducir la cantidad de tiempo que las aplicaciones conflictivas están asignadas juntas en los cores rápidos.

Por último, es necesario tener en cuenta que HSP se ve especialmente afectado por los efectos de la contención de recursos compartidos bajo ciertas cargas de trabajo como la W5 y W13-W15, en las que dos de los programas con el mayor factor de ganancia (aquellos mostrados al comienzo de cada fila en la tabla 5.1) son ambos programas muy intensivos en memoria o constituyen un par formado por una aplicación intensiva en memoria y un programa sensible a la contención. El beneficio que estas aplicaciones derivan al ejecutarse en un core rápido se produce en parte por el hecho de que este tipo de cores incorporan una mayor caché L2 compartida que los cores lentos (ver figura 2.3 para más detalles). No obstante, si el planificador asigna dos hilos intensivos en memoria a los cores rápidos de forma simultánea, ambos competirán entre sí por el espacio de la caché L2 compartida y por el ancho de banda del bus, lo cual genera una degradación significativa del rendimiento para ambas aplicaciones y empeora el rendimiento global del sistema.

5.1.2. Cargas de trabajo en la configuración 4B-4S

Procedemos ahora con la discusión de los resultados de las cargas de trabajo ejecutadas en la placa Odroid XU4 (configuración asimétrica 4B-4S), . Estas cargas se enumeran en la tabla 5.3. En esta plataforma, se trató de analizar el comportamiento de cargas de trabajo con un amplio rango de factores de ganancia o speedup factor y grados de contención variables. Cabe destacar que a la hora de construir las cargas de trabajo se tuvo que tener en cuenta el impacto agregado en el uso de memoria de las cargas de trabajo, que no debían exceder el límite de memoria física disponible en la placa Odroid XU4 (2GB) para evitar que el *Out-of-Memory killer* (OOM) de Linux entre en funcionamiento y acabe con la ejecución de las aplicaciones. Debido a esta limitación, se tuvieron que descartar algunas combinaciones de aplicaciones y benchmarks específicos para las diferentes categorías consideradas.

En general, las cargas de trabajo exploradas –mostradas en la tabla 5.3– se pueden agrupar en 3 grandes categorías. La primera combina 8 aplicaciones monohilo (M1-M8) que exhiben un diferente grado de intensidad de memoria y cubren un amplio espectro de factores de ganancia (SF). Por otra parte, las cargas de trabajo de la segunda categoría (M9-M12) se componen de 4 aplicaciones secuenciales y un programa multihilo. En este escenario, los programas secuenciales exhiben un mayor factor de ganancia que las aplicaciones multihilo cuando se ejecutan en un core rápido la mayoría del tiempo. Como se discute en la sección 4.5, debido a las necesidades de sincronización entre los diferentes hilos, cuando se asignan hilos de la misma aplicación paralela a diferentes tipos de core de forma simultánea–incluso aunque sea por períodos cortos de tiempo– se puede producir un progreso desigual y un uso ineficaz de los cores rápidos. Detectar el grado de paralelismo a nivel de

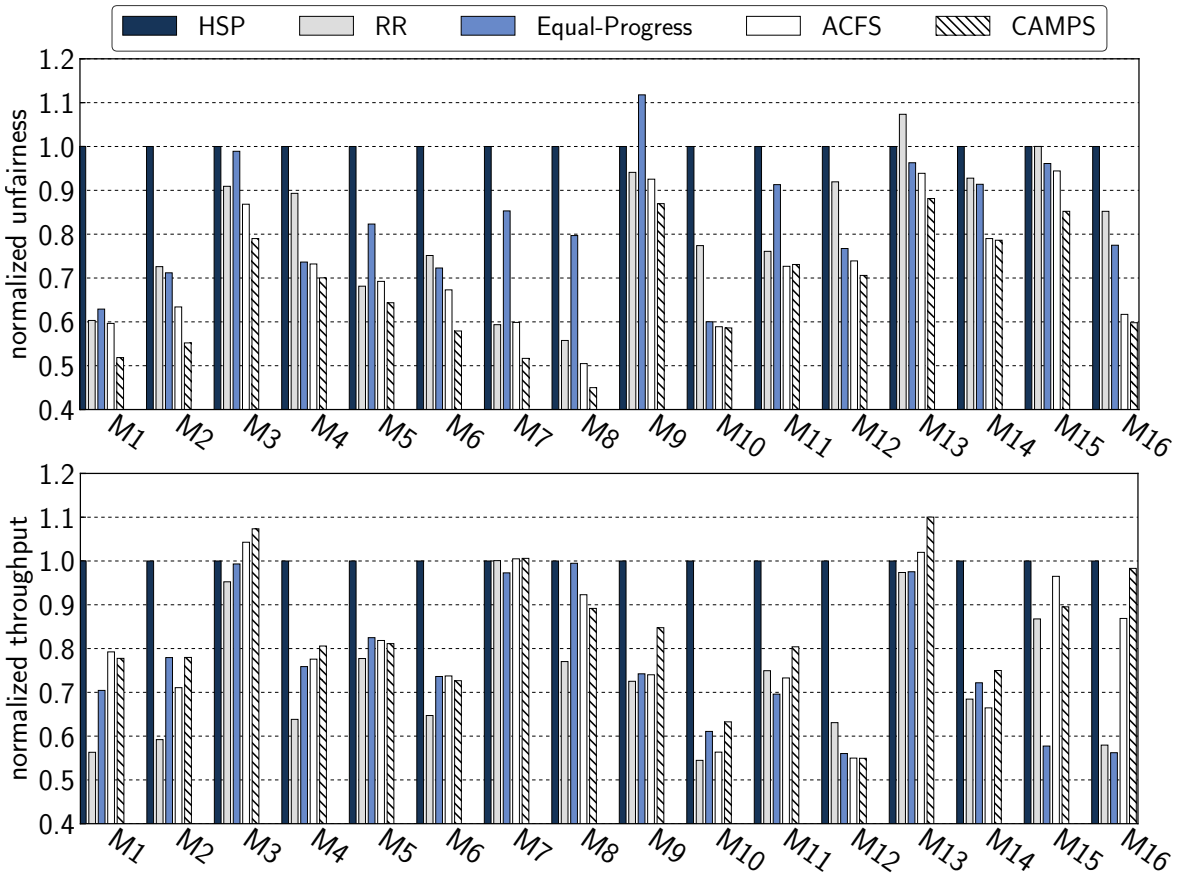


Figura 5.2: Valores de injusticia (unfairness) y rendimiento global (throughput) para las cargas de trabajo de la tabla 5.2 ejecutándose en la configuración 4B-4S bajo los distintos algoritmos de planificación

hilo (*thread-level parallelism* o TLP) de las aplicaciones es un factor clave para identificar aquellas fases de las aplicaciones que realmente se benefician de usar los cores rápidos (p.ej., las fases de ejecución secuencial) [4, 40, 39]. Finalmente, las cargas de trabajo de la tercera categoría (M13-M16) combinan dos aplicaciones paralelas con diferentes propiedades de escalabilidad. En concreto, los programas `FFTW3D`, `semphy` y `blackscholes` tienen unas fases de ejecución secuenciales que constituyen más de un 20 % de su tiempo de ejecución, mientras que `EP` y `kmeans` son aplicaciones altamente paralelas.

La figura 5.2 muestra los valores de injusticia (unfairness) y rendimiento global (throughput) de la ejecución de las cargas de trabajo de la configuración 4B-4S normalizadas con respecto a los resultados del planificador HSP. A pesar de las importantes diferencias entre las composiciones de estas cargas de trabajo y aquellas evaluadas en la configuración 2B-4S, los resultados muestran tendencias muy similares a las discutidas previamente. Esencialmente, el planificador CAMPS consigue la mayor reducción de la injusticia (hasta un 55 % en M8 relativo a HSP) para la mayor parte de cargas de trabajo. De igual manera, ACFS representa el algoritmo de planificación que ofrece las cifras de justicia más cercanas a CAMPS, seguido por Round-Robin y Equal-Progress. Además, se observa de nuevo que intentar optimizar la justicia lleva en algunos casos a degradar el rendimiento de forma significativa (hasta el 45 % en M12 relativo a HSP).

Los resultados de la tabla 5.4 indican que CAMPS todavía es capaz de alcanzar reducciones significativas de la injusticia con respecto al resto de estrategias de planificación bajo la configuración 4B-4S (32.7 % respecto a HSP, y 7 % relativo a ACFS). Nótese que, las ganancias relativas a otros algoritmos de planificación conscientes de la justicia son algo inferiores a las observadas en la configuración 2B-4S (ver tabla 5.2). Esto tiene que ver con el menor grado de intensidad de memoria de las cargas de trabajo ejecutadas en la configuración 4B-4S, que está causada por la imposibilidad (debido a la limitación de memoria a 2GB) de utilizar cargas de trabajo que combinen múltiples benchmarks altamente intensivos en memoria, los cuales suelen tener importantes requisitos de memoria.

A pesar de obtener unas mejoras limitadas de justicia en la configuración 4B-4S respecto a 2B-4S, CAMPS consigue unas mayores ganancias de rendimiento global respecto a Round-Robin y Equal-Progress en este contexto –hasta 16 % respecto a Round-Robin, y 12 % relativo a Equal-Progress. Esto sucede como resultado de una mayor diversidad de factores de ganancia presentes en las mezclas de programas, lo que se ve influenciado por dos factores. En primer lugar, el espectro de factores de ganancia es significativamente mayor en la placa Odroid XU4 (desde 1.36x hasta 6.63x) que en la placa Juno (desde 1.5x hasta 4.4x). El planificador Round-Robin no tiene en cuenta los factores de ganancia al tomar decisiones de planificación, por lo que no consigue alcanzar unas cifras decentes de rendimiento global en este

Tabla 5.4: Reducción media de la injusticia (unfairness) e incremento del rendimiento global (throughput) alcanzados por CAMPS respecto al resto de algoritmos de planificación en la placa Odroid XU4.

CAMPS vs. otros	**Reducción de injusticia**	**Incremento del rendimiento**
HSP	32.75 %	-16.03 %
RR	16.89 %	16.53 %
Equal-Progress	18.67 %	12.05 %
ACFS	7.17 %	4.51 %

contexto. En segundo lugar, algunas mezclas de aplicaciones combinan programas secuenciales, que obtienen unos beneficios no despreciables al usar un único core rápido, con aplicaciones multihilo que solo se benefician de los cores rápidos en el caso de que todos sus hilos estén asignados a este conjunto de cores de forma simultánea durante algún tiempo. En estas circunstancias, tener en cuenta el grado de paralelismo a nivel de hilo al tomar decisiones de planificación es un factor clave para mejorar el rendimiento global [4, 39] y, al mismo tiempo, dedicar los cores rápidos para ejecutar aquellas fases secuenciales con un bajo TLP produce un mayor beneficio que asignar los hilos a cores basándose únicamente en la degradación del rendimiento individual del hilo [38]. A diferencia de Round-Robin y Equal-Progress , los otros planificadores (incluyendo a CAMPS) tienen este aspecto en cuenta y reducen los factores de degradación del rendimiento o de ganancia de los hilos en función del número de hilos en estado *runnable* de la aplicación (un indicador del TLP). Al no tener en cuenta el grado de paralelismo a nivel de hilo de las aplicaciones multihilo, los planificadores Round-Robin y Equal-Progress experimentan en algunos casos una alta degradación del rendimiento global (hasta un 40 % en la M16).

5.2. Ajuste del compromiso rendimiento-justicia

Tanto CAMPS como ACFS están equipados con un parámetro configurable, conocido como *unfairness_factor* (UF), que permite al administrador del sistema obtener mejoras en el rendimiento global a costa de degradar la justicia. Cuando el parámetro *UF* se establece a su valor mínimo por defecto (1.0), ambos planificadores intentan optimizar la justicia. Por el contrario, cuando se utiliza un valor de *UF* mayor a 1, se activa una estrategia de prioridad dinámica que ajusta la prioridad de los hilos basándose en su factor de ganancia y en el *UF*.

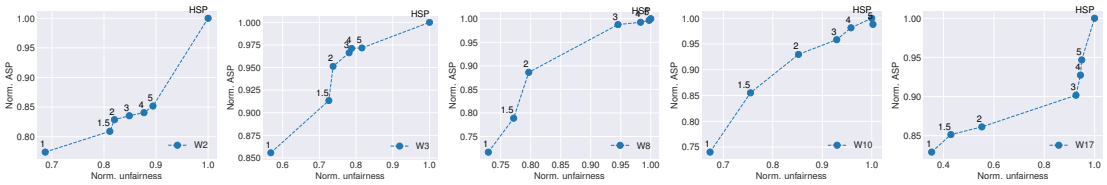


Figura 5.3: Valores normalizados de injusticia y rendimiento global para diferentes cargas de trabajo y valores del parámetro UF bajo ACFS

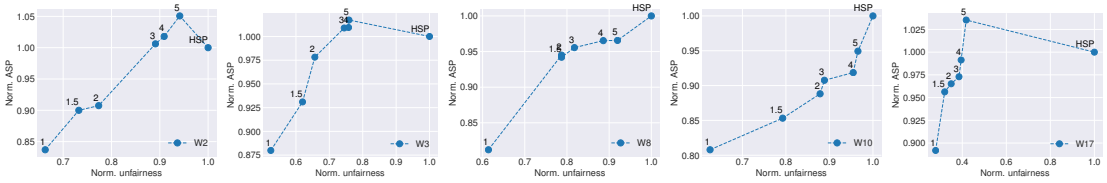


Figura 5.4: Valores normalizados de injusticia y rendimiento global para diferentes cargas de trabajo y valores del parámetro UF bajo CAMPS

En trabajos anteriores [38], se demostró la efectividad del parámetro *unfairness_factor* de ACFS en una configuración asimétrica que no estaba sujeta a los efectos de la contención en caché/bus. En este trabajo se evalúa el impacto de variar el parámetro *UF* en la justicia y el rendimiento global bajo ACFS y CAMPS en la configuración asimétrica 2B-4S, donde las aplicaciones pueden sufrir una degradación del rendimiento causada por la contención de recursos compartidos. Para este experimento se han seleccionado aquellas cargas de trabajo donde el planificador HSP obtiene un rendimiento global considerablemente superior al de ACFS o CAMPS. De esta manera, mientras se incrementa gradualmente el parámetro *UF* bajo una carga determinada, se espera encontrar que los valores de rendimiento global obtenidos por los planificadores CAMPS y ACFS se vayan aproximando a los de HSP (como se ilustra en [38] para ACFS).

En las figuras 5.3 y 5.4 se muestra cómo la elección del parámetro *UF* (en un rango entre 1 y 5)[1] afecta a la injusticia y rendimiento global del sistema para varias cargas de trabajo ejecutándose en la configuración 2B-4S bajo ACFS y CAMPS, respectivamente. Los resultados revelan que se obtienen ganancias de rendimiento global a medida que incrementamos el valor de parámetro *UF*; esto se produce también a costa de una degradación creciente de la justicia. Además, a medida que se incrementa el *UF*, los valores de rendimiento global y justicia de ambos planificadores se acercan a los del algoritmo HSP, que intenta optimizar el rendimiento global únicamente.

[1]Para la mayoría de cargas de trabajo investigadas, al incrementar el parámetro de UF más allá de 5 no se obtienen ganancias notables de rendimiento global.

También se ha observado que para algunas cargas de trabajo (W2,W3 y W6) CAMPS es capaz de mejorar ligeramente el rendimiento global respecto al planificador HSP (hasta un 5 %) cuando se usan valores altos de este parámetro. Claramente, este no es el caso de ACFS. Para comprender estos resultados, es necesario recordar que HSP y ACFS no tienen en cuenta los efectos de la contención en recursos compartidos. En las cargas de trabajo mencionadas, las aplicaciones con los mayores factores de ganancia de la carga de trabajo se encuentran también sujetas a una gran degradación del rendimiento causada por la contención. Observamos que el planificador HSP asigna estas aplicaciones sensibles a la contención a los cores rápidos simultáneamente durante mayores periodos de tiempo que CAMPS y ACFS cuando utiliza la opción por defecto de *unfairness_factor* (1.0). A medida que incrementamos el *UF*, ambos planificadores conscientes de la justicia proporcionan una mayor fracción de uso de los cores rápidos a las aplicaciones que derivan un mayor factor de ganancia (las aplicaciones sensibles a la contención en este caso). Claramente, un incremento modesto en la fracción de uso de los rápidos para estas aplicaciones intensivas en memoria produce una mejora en su rendimiento. Sin embargo, al incrementar demasiado la cantidad de tiempo que estas aplicaciones pasan asignadas simultáneamente a cores rápidos (lo que ocurre cuando se incrementa el *UF* a partir de cierto punto), provoca el efecto contraproducente de que se degrade el rendimiento individual de las aplicaciones. A diferencia de ACFS (inconsciente de la contención), CAMPS es capaz de lidiar con este problema de forma efectiva al favorecer aquellos intercambios de hilos que contribuyen a reducir la contención en el conjunto de cores rápidos. En otras palabras, CAMPS reduce la cantidad de tiempo que las aplicaciones de SF alto y sensibles a la contención pasan asignadas a cores rápidos simultáneamente. Este aspecto combinado con un alto valor del parámetro de *UF* (p.ej., 5), permite que CAMPS obtenga una mejora significativa del rendimiento global frente a HSP, un planificador inconsciente de la contención que únicamente intenta optimizar ese objetivo.

Para resumir, este estudio ha demostrado que el parámetro *unfairness_factor* permite mejorar de forma efectiva el rendimiento global del sistema en escenarios en los que la justicia no constituya un aspecto crítico. Además, al usar los valores más altos de este parámetro como en nuestros experimentos, CAMPS puede configurarse para aproximar el comportamiento del planificador HSP. En múltiples casos, CAMPS es capaz de mejorar el comportamiento de este planificador tanto en rendimiento global como en justicia. De ahí, se concluye que CAMPS constituye una estrategia de planificación consciente de la contención y versátil, ya que permite al administrador del sistema perseguir dos objetivos de optimización con un único algoritmo de planificación.

Capítulo 6

Conclusiones y trabajo futuro

A continuación se presentan las principales conclusiones obtenidas de este trabajo y tras esto, posibles avenidas de trabajo futuro.

6.1. Conclusiones

En este trabajo se ha propuesto CAMPS, un planificador a nivel del sistema operativo y consciente de la justicia para procesadores multicore asimétricos con repertorio común de instrucciones. A diferencia de otros algoritmos de planificación orientados a justicia [7, 45, 38], la aproximación utilizada tiene en cuenta de forma efectiva la degradación del rendimiento causada por la contención en recursos compartidos, como la caché de último nivel o el bus de memoria. CAMPS es capaz de registrar de forma precisa el progreso que realizan los hilos de la carga de trabajo a lo largo de su ejecución en diferentes tipos de core, y garantiza la justicia equilibrando el progreso de los hilos mediante migraciones.

En el estudio preliminar presentado en el capítulo 2 se demuestra que no es realista asumir que la degradación del rendimiento no es significativa en escenarios en los que los hilos compiten de forma intensa por el uso de recursos compartidos. Para ilustrar este hecho se experimentó en diversas plataformas AMP, lo que nos permitió obtener dos conclusiones principales:

- La degradación como consecuencia de la contención en recursos compartidos puede ser significativa (hasta 2.98x en las plataformas exploradas), por lo que deber de ser tenida en cuenta a la hora de monitorizar la degradación del rendimiento de un hilo para mantener la justicia y garantizar una utili-

zación efectiva de los cores rápidos. En realidad, para algunos benchmarks, el beneficio obtenido al ejecutarse en un core rápido respecto a uno lento podría ser reducido notablemente debido a la alta degradación del rendimiento causada por la contención de recursos compartidos.

- Monitorizar las instrucciones por segundo (IPS) de un hilo cuando se ejecuta en un core rápido en una situación de baja contención en el cluster de cores rápidos (p.ej., aquellos que comparten la LLC) podría ser una buena estimación de IPS_{alone}. Esencialmente, la penalización del rendimiento que podría sufrir un hilo asignado a un core rápido al asignar múltiples aplicaciones intensivas en memoria en los cores lentos es típicamente muy baja en comparación con la interferencia causada por los hilos intensivos en memoria que se ejecutan en cores rápidos. Esto está relacionado con la organización de la jerarquía de memoria del hardware AMP actual, así como con el hecho de que los cores lentos (con ejecución en orden) típicamente utilizan menos ancho de banda de memoria que los rápidos (ejecución fuera de orden).

El mecanismo de seguimiento del progreso se basa en la estimación de la degradación del rendimiento de un hilo en tiempo de ejecución, que se consigue comparando su rendimiento actual con el observado en el pasado cuando se ejecutó en un core rápido en una situación de baja contención. Al hacerlo, el planificador tiene en cuenta la degradación del rendimiento causada por la contención así como la ralentización que el hilo experimenta intrínsecamente al estar asignado a un core lento en lugar de uno rápido. Notablemente, la aproximación utilizada no requiere extensiones hardware especiales [44, 45] ni modelos de predicción de SF específicos de plataforma [24, 38]. En lugar de eso, CAMPS necesita monitorizar un conjunto de métricas de rendimiento para cada hilo que pueden medirse dinámicamente en multicores asimétricos comerciales mediante contadores hardware. Esto hace que el planificador sea portable entre diferentes modelos de procesador y arquitecturas.

Para evaluar la eficacia de CAMPS sobre hardware multicore asimétrico real se procedió a realizar su implementación como una clase de planificación del kernel Linux. Además, se realizó una comparativa exhaustiva del planificador propuesto con otras estrategias de planificación que intentan optimizar la justicia [7, 45, 38]. Nuestra evaluación experimental revela que CAMPS mejora la efectividad de la estrategia de planificación que constituye el estado del arte –el planificador ACFS [38]– tanto en justicia como rendimiento global (hasta un 11 % y 17 % respectivamente). Además, nuestra propuesta implementa un parámetro configurable que hace posible ajustar el grado en el que el planificador optimiza el rendimiento global o la justicia, manteniendo un único, aunque flexible, algoritmo de planificación.

6.2. Trabajo futuro

Como ya se ha demostrado en este trabajo, la contención por el uso de recursos compartidos puede afectar muy negativamente a la justicia y al rendimiento de las aplicaciones de una carga de trabajo. En este trabajo se ha seguido una aproximación software que intenta evitar la contención en sistemas multicore asimétricos mediante migraciones de hilos que minimizan el tiempo que, las aplicaciones intensivas en memoria y las sensibles a la contención, se ejecutan simultáneamente en cores rápidos. No obstante, podría resultar interesante explorar otras técnicas en sistemas simétricos basadas en asignación asimétrica de recursos utilizando el *Resource Director Technology* de Intel [34]. Esta tecnología aporta nuevos niveles de visibilidad y control sobre cómo los recursos compartidos, como la caché de último nivel (LLC) o el ancho de banda de memoria, son utilizados por aplicaciones o máquinas virtuales. Lo que podría abrir nuevas opciones para lidiar con la contención en recursos compartidos.

Por otra parte, la mayor parte de estrategias de planificación propuestas para procesadores multicore asimétricos están diseñadas principalmente para cargas de trabajo de larga duración e intensivas en cómputo, típicas de entornos HPC (*High Performance Computing*). Sin embargo, otros tipos de cargas de trabajo (aplicaciones para plataformas móviles o servidores web) más sensibles a latencia también pueden beneficiarse enormemente de la potencia y eficiencia energética de los multicore asimétricos. Sería interesante evaluar el impacto que podría tener un planificador orientado a justicia que también ofreciera soporte para este tipo de aplicaciones. Está sería sin duda una línea de investigación interesante.

Apéndice A

Introduction

Over the last years, two major trends have arisen in microprocessor design and manufacturing: the inclusion of a higher number of cores per chip, and coupling different core types on the same CPU for diverse and specific use. The second trend has given rise to growing interest in heterogeneous platforms and system layouts.

The degree of diversity divides heterogeneous architectures into various kinds, each becoming a unique point in the design space [32, 27]. One extreme of this spectrum is to couple a modest number of high-performance cores with accelerators [9] or with special-purpose processing units. This is the case of systems such as the IBM Cell Broadband Engine [19] or CPU-GPU platforms, where the different cores usually expose a different Instruction Set Architecture (ISA). Despite their benefits, these architectures typically require substantial programming effort [27, 35]. This kind of heterogeneous platforms stands in contrast with asymmetric single-ISA multicore processors (AMPs) [26], which integrate a mix of complex high-performance big cores and power-efficient small cores on the same chip. Our work targets this kind of heterogeneous platforms.

Even though applications specifically tailored to exploit the capabilities of the different cores in an AMP can be effectively run by manually binding the various threads/tasks to the core type where they are meant to run (e.g., via affinity masks), providing the potential of AMPs to unmodified applications (the ones we focus) in a transparent way poses a number of challenges to the operating system [27, 32], some of which need to be properly addressed by the scheduler [24, 38]. One important challenge is how to effectively distribute big-core cycles among the various threads running on the system. Most scheduling algorithms proposed for AMPs have been designed to optimize the system throughput for multiprogram workloads [26, 42, 24, 39, 44]. To make this happen, the scheduler must devote big cores to running those applications that derive higher performance improvements

(speedup) relative to running on small cores [26]. Further throughput gains may be obtained by using big cores to hasten sequential phases and other scalability bottlenecks present in multithreaded programs, by employing software [4, 39, 17] or hardware-aided approaches [18, 29].

Nonetheless, asymmetry-aware schedulers that seek solely to optimize throughput are known to be inherently unfair [38]. Unfairness gives rise to a number of undesirable effects on the system [33, 10]. For example, when using a scheduler that attempts to optimize throughput only in the AMP, an application's completion time may differ significantly across runs, depending on the other programs of the workload [38]. Moreover, equal-priority applications may not experiment an equal performance degradation when running simultaneously relative to the performance measured when each application isolated on the AMP. These issues make priority-based scheduling policies inapplicable [10], reduce performance predictability [27, 49, 12] and can lead to wrong billings in commercial cloud-like computing providers, where users are charged for CPU time [33].

These work primarily focuses on how to fairly schedule, at the OS level, a mix of unmodified programs on an AMP platform. With that purpose, the scheduler must even out the progress made by the different applications in the workload as they run on the various core types through their execution [45, 38]. In order to do so, the scheduler must be equipped with a mechanism that is capable of measuring the performance degradation accumulated by each application at runtime relative to its isolated execution (aka. *slowdown*). On asymmetric multicores, the performance slowdown depends on two main aspects:

- *Performance asymmetry* occurs due to the fact that most of the applications derive a significant speedup from using big cores relative to running on small ones. When the number of threads in the workload exceeds the number of big cores in the AMP, some threads may be mapped to small cores by the scheduler for a given amount of time [27, 24, 39]. When a thread runs on a small core, it slows down in proportion to its big-to-small speedup, which may differ greatly across applications and may vary over time as a program goes through different phases [26, 7].

- *Shared-resource contention* on AMPs may also lead to substantial performance degradation. In current AMP hardware, cores of the same cluster (big or small) typically share a last-level cache [5, 15, 8] and other memory-related resources. Applications running on the various cores may compete with each other for cache space and bus bandwidth, which could degrade their perfor-

mance in an uneven and unpredictable way, as the hardware itself does not capable of ensuring a fair usage of the various shared resources [46, 52, 21]. Note that DRAM-controller contention among all cores in the system may also become apparent [49, 50, 10].

Recent scheduling proposals for asymmetric multicores, such as Equal-Progress [45] or ACFS [38], attempt to enforce fairness by just taking into account performance asymmetry issues. However, they do not acknowledge shared-resource contention effects when making scheduling decisions. As we prove in this work, this leads to significant performance/fairness degradation when several memory-intensive applications are present in the workload. Conversely, contention-conscious algorithms that aim to deliver fairness [46, 12, 52] or strive to improve performance isolation [49, 48, 1] are not designed to work on systems that integrate a mix of high-performance cores and low-power cores, which may exhibit different microarchitectural features. Hence, these schemes do not take performance asymmetry aspects into account.

A.1. Project goals and work plan

To overcome the limitations of the existing proposals, this project has the goal of developing CAMPS, an OS-level contention-aware scheduler for AMPs, that strives to optimize fairness while maintaining acceptable system throughput in the context of long-running compute-intensive workloads. This work is meant to push further ahead the state of the art in fairness-oriented asymmetry-aware schedulers, by now taking shared-resource contention effects into consideration.

In order to complete this work, the following key aspects of the work plan should be dealt with.

- Perform a preliminary study in order to assess the impact of shared-resource contention on different commercial asymmetric platforms.

- Design a runtime mechanism that, based on the conclusions of the preliminary study, is capable of predicting the current slowdown that a thread in the workload experiences as it runs on the various cores of an AMP. To do so, various runtime metrics will be monitored via performance monitoring counters (PMCs).

- Implement the algorithm as a scheduling class of the Linux kernel that is capable of ensuring fairness while maintaining acceptable system throughput.

- Design of a configurable parameter that allows (1) to gradually improve throughput at the expense of degrading fairness, or (2) to optimize system throughput only.
- Perform an experimental evaluation of the proposed algorithm using real asymmetric hardware.

A.2. Master Thesis structure

The remainder of this work is structured as follows.

- **Chapter 2** introduces the notion of fairness used in this work and discusses the challenges related to measuring the slowdown of each thread at runtime. Then, it presents the results of the experimental study that assesses the effects of shared-resource contention on AMPs
- **Chapter 3** exposes how the speedup factor can be obtained at runtime and discusses different strategies that use this factor to optimize throughput. Besides, it presents different proposals that seek to guarantee fairness on AMPs. Finally, it exposes different proposals that take into account shared-resource contention.
- **Chapter 4** provides a general idea of the CAMPS implementation in the Linux kernel. Then, it describes two core components of CAMPS, the *performance monitor* and the *core scheduler*. Besides, it presents the technique used to improve the accuracy of slowdown predictions of certain threads, known as the *non-work-conserving* mode. Finally, it explains the mechanism used by CAMPS to exchange fairness for throughput.
- **Chapter 5** evaluates the effectiveness of CAMPS relative to other asymmetry-aware scheduling algorithms [7, 24, 45, 38]. Then, it shows the effects of the mechanism that allows CAMPS to trade fairness for throughput.
- **Chapter 6** presents the main conclusions obtained in this work and proposes possible avenues for future work.
- Finally, different appendices are provided: (A) Introduction and (B) Conclusions of this work translated into English.

Appendix B

Conclusions and future work

This chapter presents the main conclusions obtained in this work and proposes possible avenues for future work.

B.1. Conclusions

In this work we have presented CAMPS, an OS-level fairness-aware scheduler for asymmetric single-ISA multicores. Unlike other fairness-oriented asymmetry-aware schemes [7, 45, 38], our approach effectively takes into account the performance degradation that comes from contention on the shared resources among cores, such as the last-level cache or the memory bus. CAMPS accurately monitors the progress that the various threads in the workload make when running on the different core clusters through their execution, and enforces fairness by evening out the progress across threads.

On the preliminary study presented in the Chapter 2 we demonstrated that, in scenarios where threads contend heavily for shared resources with each other, it is unrealistic to assume that a thread's slowdown is not significant when it runs on a big core (as done in [45, 38]). To illustrate this fact we experimented with diverse AMP platforms, which allowed us to draw the two following conclusions:

1. Performance degradation due to sharing resources among big cores can be significant on current AMP hardware (up to 2.98x on the explored platforms), and should be accounted for when tracking the slowdown of a thread to enforce fairness as well as to ensure effective utilization of big cores. In fact, for some applications, the performance improvements that come from running on a big core relative to a small one could be substantially reduced due to the high slowdown that comes from shared resource contention.

2. Monitoring a thread's IPS when it runs on a big core during a low contention scenario on that cluster is typically a good estimate for IPS_{alone}. Basically, the performance degradation that threads mapped to a big core may suffer from placing multiple memory-intensive aggressors on small cores is typically very low compared to the penalty that comes from interference with memory-intensive threads running on other big cores. This is related with the memory-hierarchy organization on commercial AMP hardware and with the fact that small cores usually demand less memory bandwidth than big cores.

CAMPS's progress tracking mechanism relies on approximating the current slowdown of an application thread by comparing its actual performance with the performance observed in the past for the thread when it ran on a big core in a low contention scenario. In doing so, the scheduler factors in the contention-related performance degradation as well as the slowdown that the thread normally experiments when it is mapped to a small core rather than to a big one. Notably, our approach does not need special hardware extensions [44, 45] or platform-specific speedup-prediction models [24, 38]. Instead, CAMPS relies on the gathering of a set of performance metrics that can be easily measured online in commercial AMP hardware via performance counters. This makes the scheduler highly portable across different processor models and CPU architectures.

CAMPS was implemented as a Linux kernel scheduling class and we assessed its effectiveness on real asymmetric hardware equipped with an ARM big.LITTLE processor. An extensive comparison was performed with other existing schemes that strive to optimize fairness [7, 45, 38]. The results from the experimental evaluation reveal that CAMPS outperforms the state-of-the-art fairness-aware scheme for AMPs –the ACFS scheduler [38]– in both fairness and throughput (by up to 11% and 17% respectively). Besides, our approach implements a configurable parameter making it possible to configure the scheduler to optimize throughput or fairness, while maintaining a single, yet flexible, scheduling algorithm.

B.2. Future work

As it has been proven, the contention that comes from the use of shared resources may cause a significant degradation of the system fairness and throughput. In this work, we have proposed a software approach that strives to reduce shared-resource contention on AMPs, by means of triggering periodic migrations that minimize the time spent mapped together to big cores of memory-intensive and content-sensitive applications. Nonetheless, it could be interesting to explore other techniques based on asymmetric resource assignment using the Intel *Resource Di-*

rector Technology [34]. This tool provides new levels of visibility and control about how shared resources, such as the LLC or the memory bandwidth, are used by applications or virtual machines. Which could bring more alternatives to deal with shared-resource contention.

Furthermore, the majority of scheduling schemes proposed for AMPs are specifically tailored to deal with long-running CPU-intensive workloads that are common in HPC. However, other types of workloads more sensitive to latency (e.g., mobile applications or web servers) may also take advantage of the potential and energy efficiency of AMPs. It would be interesting to asses the impact of having a fairness-aware scheduler that offers support for this kind of applications. This could doubtlessly be an interesting avenue for future work.

Bibliografía

[1] A. Alhammad and R. Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *Proc. of RTAS'16*, pages 1–11, April 2016.

[2] T. Anderson. The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors. *IEEE TPDS*, 1(1):6–16, 1990.

[3] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs. In *Proceedings of PACT '13*, pages 63–72, 2013.

[4] M. Annavaram et al. Mitigating Amdahl's Law through EPI Throttling. In *Proceedings of ISCA 05*, pages 298–309, 2005.

[5] ARM. Juno ARM development platform. `http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.juno/index.html`, 2014. Accessed: 2017-03-09.

[6] S. Barati and H. Hoffmann. Providing fairness in heterogeneous multicores with a predictive, adaptive scheduler. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 38–49, 2016.

[7] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of CF 06*, pages 29–40, 2006.

[8] N. Chitlur et al. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proceedings of HPCA 12*, pages 1–8, 2012.

[9] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 180:1–180:6, New York, NY, USA, 2014. ACM.

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of ASPLOS 10*, pages 335–346, 2010.

[11] X. Fan, Y. Sui, and J. Xue. Contention-aware scheduling for asymmetric multicore processors. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 742–751, Dec 2015.

[12] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Perf & fair: a progress-aware scheduler to enhance performance and fairness in SMT multicores. *IEEE Transactions on Computers*, PP(99), 2016.

[13] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *Proceedings of MICRO '06*, pages 149–160, 2006.

[14] D. Gove. CPU2006 working set size. *SIGARCH Comput. Archit. News*, 35(1):90–96, Mar. 2007.

[15] Hardkernel. Odroid XU4 board. `http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4`, 2016. Accessed: 2016-6-22.

[16] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[17] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of CGO '16*, pages 24–35, 2016.

[18] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *Proceedings of ISCA 13*, pages 154–165, 2013.

[19] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, Sept 2007.

[20] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *Proceedings of ASPLOS'10*, pages 117–128, 2010.

[21] D. Kaseridis, M. F. Iqbal, and L. K. John. Cache friendliness-aware management of shared last-level caches for high performance multi-core systems. *IEEE Transactions on Computers*, 63(4):874–887, April 2014.

[22] O. Khan and S. Kundu. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, GLSVLSI '10, pages 397–400, 2010.

[23] C. Kim and J. Huh. Fairness-oriented OS scheduling support for multicore systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 29:1–29:12, 2016.

[24] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proceedings of Eurosys 10*, pages 125–138, 2010.

[25] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[26] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of ISCA 04*, pages 64–75, 2004.

[27] T. Li, P. Brett, R. Knauerhase, and D. Koufaty. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of HPCA 10*, pages 1–12, 2010.

[28] N. Markovic et al. Hardware round-robin scheduler for single-ISA asymmetric multi-core. In *Euro-Par 2015*, pages 122–134. 2015.

[29] N. Markovic et al. Thread lock section-aware scheduling on asymmetric single-ISA multi-core. *IEEE Computer Architecture Letters*, 14(2):160–163, July 2015.

[30] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[31] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[32] S. Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, 48(3):45:1–45:38, Feb. 2016.

[33] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of MICRO '07*, pages 146–160, 2007.

[34] K. Nguyen. Intel's cache monitoring technology software-visible interfaces. `https://software.intel.com/en-us/blogs/2014/12/11/intel-s-cache-monitoring-technology-software\-visible-interfaces`, 2014. Accessed: 2017-12-10.

[35] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program The Cell Broadband Engine Processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[36] M. Pricopi et al. Power-performance modeling on asymmetric multi-cores. In *Proceedings of CASES 13*, pages 15:1–15:10, 2013.

[37] J. C. Saez et al. PMCTrack: Delivering performance monitoring counter support to the OS scheduler. *The Computer Journal*, 60(1):60–85, 2017.

[38] J. C. Saez et al. Towards completely fair scheduling on asymmetric single-ISA multicore processors. *Journal of Parallel and Distributed Computing*, 102:115 – 131, 2017.

[39] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Trans. Comput. Syst.*, 30(2):6:1–6:38, Apr. 2012.

[40] J. C. Saez, A. Fedorova, M. Prieto, and H. Vegas. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the International Conference on Computing Frontiers*, pages 31–40, 2010.

[41] J. C. Saez, A. Pousa, A. E. de Giusti, and M. Prieto-Matias. On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors. *The Computer Journal*, 61(1):74–94, 2018.

[42] D. Shelepov et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *Oper. Syst. Review*, 43(2):66–75, 2009.

[43] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *Proceedings of EXADAPT '11*, pages 12–21, 2011.

[44] K. Van Craeynest et al. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of ISCA 12*, pages 213–224, 9-13 June 2012.

[45] K. Van Craeynest et al. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Procedings of PACT 13*, pages 177–187, 2013.

[46] D. Xu et al. Providing fairness on shared-memory multiprocessors via process scheduling. In *Proceedings of ACM SIGMETRICS'12*, pages 295–306, 2012.

[47] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of PACT '10*, pages 237–248, 2010.

[48] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, Nov 2016.

[49] H. Yun et al. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.

[50] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proc. of RTAS '14*, pages 155–166, 2014.

[51] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. In *Proceedings of ASPLOS 10*, pages 129–142, 2010.

[52] S. Zhuravlev et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012.

[53] S. Zhuravlev et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012.